EFFICIENT FPGA IMPLEMENTATIONS FOR HOMOMORPHIC ENCRYPTION OPERATION OF CKKS SCHEME

by
Can Ayduman

Submitted to
the Faculty of Engineering and Natural Sciences
in partial fulfillment of the requirements for the degree of
Master of Science

Sabancı University
Istanbul, Türkiye
August 2023

ABSTRACT

EFFICIENT FPGA IMPLEMENTATIONS FOR HOMOMORPHIC ENCRYPTION OPERATION OF CKKS SCHEME

CAN AYDUMAN

ELECTRONICS ENGINEERING MSC. THESIS, AUGUST 2023

Thesis Advisor: Prof. Dr. Erkay Savaş

Keywords: Homomorphic Encryption, Polynomial Multiplication, Number Theoretic Transform (NTT), FPGA, Hardware Accelerator

Homomorphic encryption is the pinnacle of cryptography, providing secure and private third-party computation of sensitive data. Homomorphic encryption schemes allow the unique ability to compute over the encrypted data. Due to its impressive power, one can gain insights from sensitive data without compromising privacy. Newer generation fully homomorphic encryption (FHE) schemes such as BFV (Fan & Vercauteren, 2012) and CKKS (Cheon, Kim, Kim & Song, 2017) schemes are the most popular and have the potential to be used in practice. The limitation of current homomorphic encryption schemes is the computationally complex operations, which prevent applications that require efficiency in their implementations. This thesis aims to present high-performance hardware designs for accelerating FHE schemes.

This thesis presents a design-time configurable hardware generator for hardware acceleration of the CKKS FHE scheme. The design aims to accelerate the multiplication, relinearization and rescale operations of the CKKS. It includes a design-time configurable Number Theoretic Transform (NTT) multiplication hardware for polynomial sizes between 2^{10} and 2^{15}. Polynomial multiplication is a bottleneck for the FHE operations. Therefore, it is crucial to design efficient hardware accelerators for high degree polynomial multiplications. The NTT enables very fast polynomial multiplication by reducing its complexity to $\mathcal{O}(n \log_2 n)$ from $\mathcal{O}(n^2)$. The Forward NTT operations are implemented with Cooley-Tukey, while Inverse NTT operations are implemented with Gentleman-Sande butterfly circuits.

Memory access pattern (MAP) of the NTT operation is complex and it is crucial to design an efficient MAP for NTT for implementing a high-throughput NTT architecture. We designed and implemented an efficient algorithm for the MAP of NTT and generalized this approach for polynomial sizes, 2^{10} to 2^{15}. Our hardware acceleration for the CKKS fully homomorphic encryption scheme offers a 15-fold speedup in the homomorphic multiplication operation and a 4-fold speedup in the key-switch operation compared to the Microsoft SEAL library. This comparison was conducted in an environment where the software ran on an AMD Ryzen 7 3800x CPU.

ÖZET

CKKS ŞEMASININ HOMOMORFİK ŞİFRELEME İŞLEMLERİ İÇİN VERİMLİ FPGA UYGULAMALARI

CAN AYDUMAN

ELEKTRONİK MÜHENDİSLİĞİ YÜKSEK LİSANS TEZİ, AĞUSTOS 2023

Tez Danışmanı: Prof. Dr. Erkay Savaş

Anahtar Kelimeler: Homomorfik Şifreleme, Polinom Çarpması, Sayılar Teorisi Dönüşümü (NTT), FPGA, Hızlandırıcı Donanım

Homomorfik şifreleme, hassas verilerin güvenli ve mahremiyet korumalı bir şekilde işlenebilmesini sağladığı için kriptografi biliminin zirvesidir. Homomorfik şifreleme şemaları, şifrelenmiş veriler üzerinde hesaplama yapabilme yeteneğine sahiptir. Bu benzersiz yeteneği sayesinde, gizliliği ve mahremiyeti tehlikeye atmadan hassas verilerden bilgi elde edilebilir. Yeni nesil tam homomorfik şifreleme (FHE) şemaları olan BFV (Fan & Vercauteren, 2012) ve CKKS (Cheon, Kim, Kim & Song, 2017) en popüler olanlardır ve pratikte kullanılma potansiyeline sahiptir. Mevcut homomorfik şifreleme şemalarının uygulamada kullanılmalarının önündeki en önemli engel ve sınırlama, bu şifreleme şemalarının bünyelerinde karmaşık işlemler barındırması ve büyük hacimli veri yapılarıyla çalışmaları nedeniyle çok yüksek ve kaynak gereksinimine ihtiyaç duymalarıdır. Bu tezin amacı, tam homomorfik şifreleme şemalarını hızlandırmak için yüksek performanslı donanım tasarımlarını sunmaktır.

Bu tez, CKKS tam Homomorfik şifreleme şemasının donanım hızlandırması için tasarım zamanında yapılandırılabilir bir donanım üreteci sunmaktadır. Tasarım, CKKS'nin çarpma, yeniden doğrusallaştırma ve yeniden ölçeklendirme işlemlerini hızlandırmayı amaçlamaktadır. 2^{10} ve 2^{15} arasında polinom dereceleri için tasarım zamanında yapılandırılabilir bir Sayısal Teorik Dönüşüm (NUmber Theoretic Transform - NTT) çarpma donanımı içermektedir.

Polinom çarpımı, FHE işlemleri için bir darboğaz oluşturmaktadır. Bu nedenle, bu işlemler için verimli donanım hızlandırıcılar tasarlamak kritiktir. NTT, karmaşıklığını $\mathcal{O}(n^2)$'den $\mathcal{O}(n \log_2 n)$'ye indirerek çok hızlı polinom çarpımını mümkün kılar. İleri NTT işlemleri Cooley-Tukey ile, ters NTT işlemleri ise Gentleman-Sande kelebek devreleri ile gerçekleştirilmektedir.

NTT işleminin Bellek Erişim Paterni (MAP) karmaşıktır ve yüksek verimlilikli bir NTT mimarisi uygulamak için verimli bir MAP için NTT tasarlamak kritiktir. NTT'nin MAP'ı

için verimli bir algoritma tasarlanmış ve uygulanmıştır, ve bu yaklaşım 2^{10} ile 2^{15} arasındaki polinom boyutları için genelleştirilmiştir. CKKS tam Homomorfik şifreleme şemasının hızlandırması için yaptığımız donanım, homomorfik çarpma işleminde Microsoft SEAL kütüphanesine kıyasla 15 kat, anahtar değiştirme işleminde ise 4 kat hız artışı sunmaktadır. Bu karşılaştırma, yazılımın AMD Ryzen 7 3800x CPU'da çalıştırıldığı bir ortamda gerçekleştirilmiştir.

ACKNOWLEDGMENT

First and foremost, I sincerely thank my supervisor, Prof. Dr. Erkay Savaş, for guiding me throughout my master's thesis journey, supporting me with his extensive knowledge, and always being by my side. Prof. Dr. Savaş's mentorship has been instrumental in forming and completing this thesis. His perfectionism, attention to detail, and meticulous approach to scientific research have consistently pushed me to achieve higher standards at every stage of my thesis work. This pursuit of excellence has been a significant source of motivation in enhancing the quality of my thesis.

Similarly, I am profoundly grateful to my ex-supervisor, Dr. Erdinç Öztürk, for his critical feedback, scientific approach, and exceptional guidance throughout my master's thesis journey. Dr. Öztürk's consistently supportive attitude, valuable suggestions, and direction have significantly influenced the shaping of my thesis. He was always by my side during my challenges, providing excellent support. Working with him has left a positive and lasting impact on my academic and personal development.

I want to express my deep gratitude to Prof. Dr. Sıddıka Berna Örs Yalçın and Prof. Dr. Ferruh Özbudak, who served on the jury committee. Their comments and reviews have greatly assisted in enhancing the quality of my thesis.

I am deeply grateful to the Cryptography and Information Security Group (CISEC) at Sabanci University. The support from the group has played a significant role in the success of my thesis. I want to thank Kemal Derya and Ahmetcan Mert from this group. Their valuable support and sharing of their experiences have contributed significantly to developing my thesis work.

I owe a deep debt of gratitude to my friends Emre and Selim for the energy they infused in me to overcome challenges during my thesis journey. They stood by my side academically and personally during the limitations I faced and in stressful moments. Their support transformed this journey from a scientific effort into a personal growth and learning experience. I thank them profoundly for their invaluable support throughout this process.

Lastly, I would like to sincerely thank my family, who have been by my side with their belief, love, and support from the beginning to the end of this process. The faith of my mother, Nurcan, and my father, Şakir, in me has been a source of strength during my challenging times and my greatest motivation in achieving success.

This thesis is partially supported by the European Union's Horizon Europe research and innovation programme under grant agreement No: 101079319 and by TUBITAK under Grant Number 118E72.

To my beloved family...

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

Fully Homomorphic Encryption (FHE) is a form of encryption that allows for computations to be performed on encrypted data without decrypting it. This property was first demonstrated as a result of the research by Gentry et al. as presented in Gentry (2009). By using FHE, when decrypted, calculations conducted on encrypted data produce the same results as those on unencrypted data. The main objective of FHE is to permit computations in the encrypted domain, ensuring the confidentiality of the data during these computations. The significance of FHE stems from its capability to maintain the privacy and security of sensitive information, particularly in cloud computing and multi-party computation scenarios. It offers a secure approach to processing confidential data without accessing the unencrypted form.

The Cheon-Kim-Kim-Song (CKKS) scheme Cheon (2017) is pivotal for conducting non-linear operations on real numbers. Originally, CKKS started resembling other Somewhat Homomorphic Encryption (SHE) systems before transitioning into a Fully Homomorphic framework. SHE systems support a finite set of arithmetic operations on encrypted data until the associated 'noise' surpasses a specific limit. Once this noise budget is depleted, further operations might result in decryption inaccuracies. On the other hand, a typical FHE system facilitates endless computations. The progression from SHE to FHE often hinges on a technique termed *bootstrapping*, introduced by Gentry Gentry (2009). While this approach is advantageous, it demands significant computational effort and serves to reset the ciphertext's noise level, allowing for sustained computations. However, *bootstrapping* is beyond the scope of this work.

Microsoft SEAL (2020) is a software library designed to implement FHE operations, offering building blocks for application development. Among the implemented HE schemes in

Algorithm 1 Algorithm of Schoolbook Multiplication

Input: $A(x), B(x) \in R_{q,n}$
Output: $C(x) = A(x) \times B(x) \in R_{q,2n}$
1: **for** i from 0 by 1 to $n-1$ **do**
2: **for** j from 0 by 1 to $n-1$ **do**
3: $C[i+j] \leftarrow (C[i+j] + (A[i] \times B[j]))modq$
4: **return** $C(x)$

Algorithm 2 Algorithm of Karatsuba Multiplication

Input: $A(x) = A_1 \cdot x^{n/2} + A_0$ where $A_1, A_0 \in R_{q,n/2}$
Input: $B(x) = B_1 \cdot x^{n/2} + B_0$ where $B_1, B_0 \in R_{q,n/2}$
Output: $C(x) = A(x) \times B(x) \in R_{q,2n}$
1: $V_0(x) = \text{Karatsuba}(A_0(x)B_0(x))$ # *Recursive call*
2: $V_1(x) = \text{Karatsuba}((A_0(x) + A_1(x))(B_0(x) + B_1(x)))$ # *Recursive call*
3: $V_2(x) = \text{Karatsuba}(A_1(x)B_1(x))$ # *Recursive call*
4: $C(x) = V_2(x) \times x^n + (V_1(x) - V_0(x) - V_2(x)) \times x^{n/2} + V_0(x)$
5: **return** $C(x)$

Microsoft SEAL is CKKS, which is particularly suitable to hardware acceleration because of its computationally intensive operations, amenable to massive parallelization.

In HE schemes, the bottleneck in terms of computational time often revolves around the multiplication of polynomials. For these encryption systems to be practically useful, it is imperative to incorporate fast and efficient methods for performing these polynomial multiplications. Various algorithms such as the traditional schoolbook multiplication, the more optimized Karatsuba algorithm, and multiplication based on the Number Theoretic Transform (NTT) can accelerate the homomorphic operations of the CKKS scheme .

Two common algorithms are often considered in polynomial multiplication: Schoolbook and Karatsuba (Karatsuba and Ofman (1962)). The steps of these two algorithms are respectively given in Algorithm 1 and Algorithm 2. Both algorithms take as inputs two polynomials $A(x)$ and $B(x)$ with degrees of $n-1$, represented as arrays of coefficients. The Schoolbook algorithm is a straightforward approach that initializes a zero-filled array $C(x)$ of size $2n-1$ to store the result. It iteratively multiplies each coefficient of $A(x)$ by each coefficient of $B(x)$, accumulating the results in $C(x)$, and has a time complexity of $\mathcal{O}(n^2)$. This complexity makes it suitable for small to moderately high degree polynomials. On the other hand, the Karatsuba algorithm divides each input polynomial into two half-degree polynomials. It uses a clever trick to reduce the number of multiplications to three: $V_0(x)$, $V_1(x)$, and $V_2(x)$. These multiplications are performed recursively, which enhances the algorithm's efficiency for higher-degree polynomials. The final result $C(x)$ is then composed of these partial products, and the algorithm has a time complexity of

$\mathcal{O}(n^{\log_2 3})$. The Karatsuba method is generally faster than the Schoolbook algorithm for larger polynomials, making it more suitable for larger-scale tasks.

In the context of Homomorphic Encryption (HE), polynomial multiplication is often required to be performed on polynomials with large degrees, such as 8192 and higher. Due to the high computational overhead associated with standard multiplication algorithms such as Schoolbook and Karatsuba, the polynomial multiplication algorithm based on NTT is often employed. NTT is a type of discrete Fourier transform (DFT) in a finite field and is an efficient algorithm that offers significant performance improvements for polynomial multiplication. Particularly in finite fields that are friendly to modular arithmetic, NTT can achieve multiplication with a time complexity of $\mathcal{O}(n \log n)$, making it highly effective for large-scale polynomial multiplications required in HE.

The SEAL library's CKKS scheme supports various parameter sets. The choice of parameter sets directly affects the balance between performance and security, computational complexity, and error rates. Therefore, it is important that the hardware can be easily configured to accommodate different parameter sets. Furthermore, the balance between performance and resource utilization varies depending on the platform and application. At this point, we want to emphasize the critical role of design-time flexible hardware. A flexible hardware design can be easily configured to be used with various parameter sets, thereby becoming suitable for different performance and security needs as well as noise budget in SHE. In other words, the same hardware infrastructure can be optimized for a large number of various applications and platforms. Such flexibility provides a significant advantage, especially in limited resources or complex systems with diverse workloads and data types. As a result, the adaptability offered by design-time flexible hardware can further broaden the practical applications of HE, thereby enhancing data privacy and security on a more comprehensive scale.

The primary motivation of this thesis is to overcome the computational challenges that hinder the broad-scale applicability of HE, thereby enabling enhanced data privacy across a broader range of applications. To this end, we emphasize the importance of hardware accelerator capable of performing large-degree polynomial multiplications efficiently and effectively, an area where many existing algorithms fall short. This accelerator is specifically optimized for NTT operations, which play a critical role in the CKKS scheme. Notably, the crucial importance of design-time flexible hardware becomes evident. Such flexibility allows the hardware to adapt quickly to various parameter sets, meeting different performance and security requirements. This adaptability can further broaden the practical applications of HE, providing more comprehensive data privacy and security. In summary, this thesis introduces a new hardware approach capable of overcoming

computational challenges and potentially expanding the practical applications of HE.

1.1 Prior Work

In previous works, there are various implementations for the NTT and CKKS schemes; however, these implementations generally focus on specific ring sizes and numbers of processing elements (PEs), which is a hardware unit responsible essentially for executing the most fundamental operation of NTT, known as butterfly operation. Such approaches need more flexibility to accommodate a wide range of polynomial degrees or different configurations of PEs. Existing works are often tailored to specific applications or performance requirements, limiting their adaptability for different scenarios or variable sizes. Therefore, we present architectures from other works that target various polynomial sizes to ensure a fair comparison. As these architectures are generally optimized in line with specific design requirements and performance objectives, making direct comparisons between them is challenging.

Work	**Platform**	$\log_2(n)$	**q**	**# of PE**
Ozturk et al. (2017)	Virtex-7	14	32-bit	Fixed
Sinha Roy et al. (2019)	UltraScale	12	30-bit	Fixed
Matteo et al. (2023)	Zynq Ultrascale+	Configurable	32-bit	Fixed
Duong-Ngoc et al. (2022)	Zynq UltraScale+	16	60-bit	Fixed
Paludo and Sousa (2022)	Virtex Ultrascale+	Configurable	32-bit	Configurable
Su et al. (2022)	Virtex-7	Configurable	32-bit	Configurable
Ye et al. (2022)	Virtex-7	Configurable	60-bit	Fixed
Xin et al. (2021)	Zynq UltraScale+	12	40-bit	Fixed
Our Work	Alveo U280	Configurable	32-bit	Configurable

Table 1.1 Comparative Table for Merged NTT Implementations in the literature.

In a previous study by Ozturk et al. (2017), the authors designed and implemented a polynomial multiplier for HE schemes using a Virtex 7 XC7VX690T FPGA. Their architecture supports polynomials of degree $N = 16,384$ and $N = 32,768$ for homomorphically computing the operations of blocks ciphers Prince and AES, respectively, with coefficient bit-lengths of $\log p = 32$ bits. Interestingly, the parameter (i.e., ciphertext modulus) q has a bit-length of 500 bits for the Prince scheme. The authors also used the Chinese Remainder Theorem (CRT) to accelerate computation and improve performance.

Although the work illustrates the feasibility of handling large-degree polynomials and incorporating CRT for optimization, it does not delve into design scalability or resource constraints issues. The work is valuable for understanding FPGA-based implementations of lattice-based cryptosystems and provides insights into the complexities of scaling such architectures.

In the study by Sinha Roy et al. (2019), the authors present a specifically optimized FPGA-based architecture to accelerate HE computations. They significantly improve memory access patterns and computational resources by utilizing the Zynq FPGA platform. Their architecture employs two parallel *butterfly* cores for NTT computations. This allows them to achieve performance close to the optimal memory access speed of eight coefficients per cycle provided by the FPGA's Block RAM (BRAM). However, the study does not offer a scalable architecture; it is only optimized for a specific ring size and modulus value.

A recent study by Matteo et al. (2023) presents a hardware accelerator specifically for the SEAL-Embedded library. This accelerator has a configurable NTT unit that can handle various polynomial degrees. However, the study is optimized for specific ring dimension and size of the modulus q and does not offer a scalable architecture. For instance, in the indicated FPGA hardware setup, the counts of LUTs and REGs increase for different configurations with polynomial degrees ranging from 1024 to 16384, while the number of DSPs remains constant. We aim to investigate as to how performance will be affected for different polynomial degrees and modulus size. This is the gap in the literature we intend to address in this work.

Duong-Ngoc et al. (2022) proposes an area-efficient NTT architecture for HE technology. At the core of the proposed NTT architecture is a high-throughput butterfly unit array that communicates with a single data memory unit through a conflict-free memory access pattern. In addition, a twiddle factor generator has been developed to reduce memory consumption. The proposed architecture has been implemented on the Xilinx Zynq UltraScale+ ZCU102 FPGA platform. The design specifically focuses on a ring size of 65536 and a certain number of processing elements. The focus on a specific set of parameters aims to enhance efficiency and performance in particular application scenarios.

Paludo and Sousa (2022) introduces accelerated architectures for NTT designed explicitly for FPGAs, utilizing a novel Montgomery-based butterfly structure. The butterfly architecture is expanded into a Modular Arithmetic Logic Unit (MALU) to enhance reuse and simplify programmability. Furthermore, to boost performance, a Linux-compatible RISC-V core with six-stage pipeline is extended with custom instructions. The per-

formance of the proposed architectures is evaluated on the Xilinx Ultrascale+ FPGA platform.

Su et al. (2022) presents a specialized NTT/INTT architecture specifically designed to enhance the performance of FHE applications. The work proposes an area-efficient and highly unified reconfigurable architecture with a variable number of processing elements, significantly boosting the computational efficiency of FHE. Implemented on a Xilinx Virtex-7 FPGA platform, the realized architecture demonstrates substantial improvements in NTT/INTT operations and polynomial multiplication compared to previous works, making the practical applications of FHE more accessible.

Ye et al. (2022) proposes an optimized pipelined NTT architecture on FPGA to accelerate polynomial multiplication. The proposed architecture offers high efficiency using reduced hardware resources and significantly improves performance, particularly for Post-Quantum Cryptography (PQC) applications.

Xin et al. (2021) offers a multi-level parallel hardware accelerator for homomorphic computations in machine learning. The vectorized NTT unit and the Residue Number System (RNS) are employed to achieve low and mid-level parallelism, respectively. Additionally, a fully pipelined and parallel accelerator is designed for dual ciphertext processing, offering high-level parallelism.

Work	**Platform**	$\log_2(n)$	**q**	**# of PE**
Mert et al. (2022)	Alveo U250	14(Configurable)	438-bit	Configurable
Riazi et al. (2020)	Stratix10	13(Configurable)	218-bit	Configurable
Our Work	Alveo U280	13	218-bit	Configurable

Table 1.2 Comparative Table for CKKS Implementations in the literature.

In the paper by Mert et al. (2022), the authors introduce "Medha", a programmable hardware accelerator, specifically designed to speed up HE algorithms for cloud-based computations. Medha features a flexible architecture that can accommodate multiple sets of HE parameters. The authors propose a "divide-and-conquer" technique, allowing the hardware accelerator designed for a smaller ring to operate on a larger ring size as well. Medha's architecture comprises interconnected parallel processing units that require a minimum number of nets, making the placement and routing of the architecture easy on the reconfigurable hardware platform. The study was implemented on a Xilinx Alveo U250 FPGA and achieved significant speed gains compared to the Microsoft SEAL software implementation. This accelerator aims to substantially improve the performance of HE, which is both computation and data-centric.

The paper by Riazi et al. (2020) focuses on the challenges of data privacy, security, and confidentiality that arise with the rapid growth of cloud computing. The study introduces a novel hardware architecture for FHE aimed at reducing the significant computational overhead, which is the main hindrance to the large-scale deployment of FHE. Utilizing multiple levels of parallelism, the architecture achieves substantial performance improvements. Specifically, the paper presents a highly parallelizable architecture for NTT, which could be of independent interest as NTT is frequently utilized in many lattice-based cryptography systems. Implementations on reconfigurable hardware demonstrate a 164-268× performance improvement across a wide range of FHE parameters. Detailed comparisons and the performance of these two designs in terms of resource utilization and performance matrix are provided in Chapter 4.

1.2 Our Contribution

Previous studies have generally focused on specific ring sizes and numbers of PEs, which means they lack the flexibility to accommodate different polynomial sizes or PE configurations. Specifically, existing studies are usually tailored to specific applications or performance requirements, limiting their adaptability to different scenarios or variable sizes. In our study, we propose a design-time configurable, parametric unified NTT hardware architecture that can meet different ring sizes, hardware resource constraints, and numbers of PEs. This flexible structure provides the capability to use the architecture for various hardware environments, polynomial sizes, and numbers of PEs. Additionally, we propose and implement a design-time flexible hardware architecture specifically optimized for NTT operations, featuring a Memory Access Pattern (MAP) method and NTT architecture. These improvements offer significant advantages in speed and scalability.

Mainly, we apply this architecture to the key-switching and homomorphic multiplication operations of the CKKS scheme, providing design-time reconfigurability by enabling an increase in the number of PEs. As a result of our hardware optimizations, we achieve a 15-fold speed-up in homomorphic multiplication operations and a 4-fold speed-up in key-switching operations compared to Microsoft SEAL's CPU implementation. These performance gains demonstrate the value of our architecture, not just in theory but also in practical applications. This thesis further extends the implementation of the CKKS scheme for varying numbers of PEs, building upon the initial work detailed in our previous

publication (Ayduman et al. (2023)), where the scheme was implemented with a fixed number of PEs.

Our work's key contributions are outlined below:

- We propose a parametric merged-NTT hardware architecture that is capable of accommodating various ring sizes and hardware resource constraints. Our thesis will detail how this flexibility allows the architecture to be employed across diverse hardware environments and ring sizes, meeting different computational and resource requirements.
- We propose an optimized MAP tailored explicitly for NTT operations. In the thesis, we aim to investigate the extendability or customizability of this efficient MAP algorithm.
- We also introduce a design-time flexible hardware architecture for the four-step NTT method, which serves as an alternative algorithm for the rapid computation of NTT. In our performance tests for specific ring sizes used in CKKS operations, we observed that the merged NTT exhibited superior performance compared to the four-step NTT. However, it's crucial to emphasize the potential of the four-step NTT. This method might offer superior performance, especially for larger ring sizes, and may be more effective in specific application scenarios. In our thesis, we will explore the advantages of this design-time flexibility for this particular NTT method and its potential contributions, especially for applications operating at higher ring sizes.
- We propose a design-time configurable hardware architecture that accelerates the multiplication and key-switching operations of the CKKS scheme, showcasing a 15× and 4× speed-up for homomorphic multiplication and key-switching operations, respectively, compared to Microsoft SEAL's CPU implementation. In the context of our thesis, we will explore the scalability and optimization of this architecture for different application scenarios and discuss how these performance gains could be translated into real-world applications.

2. BACKGROUND

This chapter introduces the theoretical and technical concepts necessary to follow and understand discussions in the thesis. These concepts of background contribute to better comprehension of the research methodology and findings discussed later in this work. In particular, mathematical notations and symbols that will be used throughout the study are introduced to set the stage. Concepts such as polynomial multiplication and NTT directly affect the functioning and performance of HE. Additionally, we will provide a detailed examination of two significant contributions in this field: the Cheon-Kim-Kim-Son Homomorphic Encryption Scheme and Microsoft SEAL Homomorphic Encryption Library. Lastly, we will discuss the Residue Number System and its impact on the performance of HE. All these topics form the foundation for the analyses and discussions that will follow in subsequent parts of the thesis.

2.1 Notation

The value q represents the modulus, with $\mathbb{Z}_q$ comprising all positive integers smaller than the modulus q. The polynomial $\Phi(x)$ denotes a cyclotomic polynomial, while $\mathbb{Z}_q[x]/\Phi_m(x)$ specifies the ring of polynomials $\mathbb{Z}_q[x]$ reduced by the polynomial $\Phi_m(x)$. The constant ω is the n-th root of unity and ψ is the $2n$-th root of unity in $\mathbb{Z}_q$. Here, n designates the degree of the cyclotomic polynomial, Φ_m (usually $\Phi_m = x^n + 1$ and $m = 2n$) and PE represents the total count of processing units, also known as butterfly units. The constant R is called the Montgomery constant, which is used to transform an

integer to its Montgomery representation when Montgomery reduction algorithm is used for modular arithmetic. Generally, $R = 2^{\ell}$ where $\ell = \log q$ for a modulus q.

The function `BitReverse(value, bit length)` returns the given binary value in bit reverse order; for instance, `BitReverse(`11_2`,4)` yields the value 1100_2. The symbol $\odot$ stands for coefficient-wise multiplication. $\bar{A} = \text{NTT}(A)$ represents the application of the NTT operation to the vector A, implying that the output resides in the NTT domain. Conversely, $\text{INTT}(\bar{A})$ indicates the application of the inverse NTT operation to the vector $\bar{A}$ in the NTT domain. The function `modinv`(n, q) is employed to compute the modular inverse of a number, aiming to determine the inverse of a number with respect to a specified modulus using the Extended Euclidean Algorithm. The function `generateTwiddleFactorTable`(w, n, q) is utilized to produce a "twiddle factor" table specifically for the NTT steps of the 4-step NTT process. This function accepts three parameters: the base twiddle factor (i.e., primitive root of unity), the size of the twiddle factor table, and the modulus value in use. In its operation, the function computes the bit-reverse of a certain value i, which is then used as the power of the base twiddle factor. The size of the table produced by this function corresponds to the number of rows or columns of the matrix being used. On the other hand, the `generateTwiddleFactorMainTable` function is tailored for the multiplication step of the Four-Step NTT process. As opposed to the former, this function generates a "twiddle factor" table with a size equal to n. It essentially builds a more expansive table to accommodate the requirements of the multiplication step. The critical distinction between the two functions lies in their purpose and the dimensions of the tables they produce. While both are integral to the 4-step NTT, they cater to different steps.

2.2 NTT-based Polynomial Multiplication

In the earliest conceptual stages of NTT, it was fundamentally intertwined with the development and understanding of Fourier transforms, specifically the Fast Fourier Transform (FFT). The foundational grounds for NTT were laid during the mid-20th century, around the same time that Cooley and Tukey introduced the FFT algorithm in Cooley and Tukey (1965), a method that significantly enhanced the efficiency of Fourier transforms by reducing the number of operations needed.

Despite the efficacy of the FFT, there was an understanding that operating over real or

complex numbers can introduce numerical instability due to round-off errors, sparking interest in transforms over finite fields or rings, which would be immune to such errors. This marked the inception of NTT, a transform that operates over finite rings, which promised computational stability and accuracy.

Early works venturing into NTT often revolved around identifying suitable rings and understanding the characteristics that these rings must possess to facilitate a successful transformation. Studies from the 1970s, such as those by Pollard (1971) and Agarwal and Burrus (1975), delved deep into the mathematical intricacies and potential of NTT. These pioneering works set the stage for the impressive development trajectory NTT would follow in the subsequent years.

As the computational power increased and the requirements of various applications became more demanding, the adaptations and implementations of NTT have evolved significantly. Nowadays, it forms a core part of different sophisticated algorithms in the field of cryptography, such as HE and lattice-based cryptographic systems, ensuring security and efficiency. Moreover, researchers have endeavored to optimize NTT further, creating variations with improved computational speed and reduced memory footprint, as demonstrated by Aysu et al. (2013). Additionally, the NTT found its place in the digital signal processing realm, enhancing the efficiency of several applications. Current adaptations leverage the potent combination of hardware accelerators and optimized algorithms to realize the full potential of NTT in modern computing environments.

In recent years, the application of NTT in the field of cryptography has garnered substantial attention. Leveraging the strengths of finite field arithmetic, NTT facilitates secure and efficient cryptographic protocols. Its integration in cryptographic operations, such as HE, allows for secure data processing while maintaining confidentiality. Besides, NTT finds crucial applications in lattice-based cryptography, which is anticipated to play a pivotal role in the post-quantum cryptography era, offering resistance against quantum attacks, as noted by Li et al. (2023). Furthermore, it aids in the swift execution of polynomial multiplication, a fundamental operation in many cryptographic systems, enhancing system performance and security. As cryptographic techniques become more advanced, the role of NTT is expected to expand, cementing its position as an indispensable tool in fostering secure communications in the digital age.

NTT is a version of the Discrete Fourier Transform (DFT) defined over the ring $\mathbb{Z}_q$. This transformation facilitates the conversion of any vector with n elements in the polynomial domain (i.e., The coefficients of polynomials of degree $n-1$) to another vector with n elements in the NTT domain. This transformation process utilizes a parameter known as

the "twiddle factor" (or primitive root of unity), denoted as ω. The following equations mathematically define the forward (NTT) and inverse (INTT) transformations:

$$\bar{a}_i = \sum_{j=0}^{n-1} a_j \omega^{i \times j} \mod q \text{ for } i = 0, 1, \ldots, n-1, \tag{2.1}$$

$$a_i = \frac{1}{n} \sum_{j=0}^{n-1} \bar{a}_j \omega^{-i \times j} \mod q \text{ for } i = 0, 1, \ldots, n-1. \tag{2.2}$$

These equations outline the mathematical operations required to find a vector's counterpart in the NTT domain.

The powers of this constant value, referred to as the "twiddle factor", play a critical role in the NTT and inverse NTT (INTT) computations. Within this scope, selecting two specific twiddle factors that satisfy certain mathematical criteria is essential. The first factor, denoted as ω, is defined as the n-th root of unity in $\mathbb{Z}_q$ and meets the conditions $\omega^n \equiv 1 \pmod{q}$ and $\omega^i \neq 1 \pmod{q}$ for all $i < n$, where q $\equiv 1 \pmod{n}$. The second factor, denoted as ψ, is designated as the $2n$-th root of unity in $\mathbb{Z}_q$, fulfilling the conditions $\psi^{2n} \equiv 1 \pmod{q}$ and $\psi^i \neq 1 \pmod{q}$ for all $i < 2n$, where $q \equiv 1 \pmod{2n}$. These two factors are interrelated through the equations $\omega = \psi^2 \bmod q$ and $\psi^n \bmod q = -1$, which are vital in ensuring the accuracy of the computations.

Based on the principles outlined in Winkler (1996), the multiplication of two $(n-1)$-degree polynomials results in a $(2n-2)$-degree polynomial. To efficiently compute this using NTT, both input polynomials are initially extended to $2n$-element vectors through zero-padding. Subsequently, a $2n$-point NTT is applied to these extended polynomials, followed by a coefficient-wise multiplication. The result is then transformed to the polynomial domain using a $2n$-point INTT. The final step involves reducing this result modulo a cyclotomic polynomial, $\Phi(x)$, to obtain the desired n-degree polynomial product. Mathematically, this process can be represented as:

$$C(x) = \text{INTT}\left(\text{NTT}(A(x)) \cdot \text{NTT}(B(x))\right) \mod \Phi(x), \tag{2.3}$$

where:

- $A(x)$ and $B(x)$ represent the zero-padded $2n$-element input vectors comprised of the coefficients of these two polynomials.
- $\text{NTT}(A(x))$ and $\text{NTT}(B(x))$ denote the transformations of these polynomials in the NTT domain.
- $C(x)$ is the output polynomial.
- $\Phi(x)$ is the cyclotomic polynomial utilized in the reduction step.

As highlighted by Pöppelmann and Güneysu (2012), utilizing positive wrapped convolution (PWC) and negative wrapped convolution (NWC) techniques enables these operations to be executed more efficiently, utilizing n-point NTT and INTT, thanks to the reduction with the cyclotomic polynomial. This optimization reduces memory usage and accelerates the computation because it facilitates operations over polynomials with lower degrees.

The PWC technique is employed when the polynomial takes the form $x^n - 1$. This method leverages the properties of cyclotomic polynomials to simplify convolution operations. Instead of expanding the convolution to broader polynomials, PWC confines the operations to positive indices.

On the other hand, the NWC method is employed when the polynomial takes the form $x^n + 1$. In this method, there is an additional step before NTT and after the INTT operation. The NTTs of the two input polynomials are taken (each being element-wise multiplied by a $\boldsymbol{\psi}$ vector, which consists of successive powers of ψ and then a coefficient-wise multiplication is carried out on them. Subsequently, these outputs are transformed back using INTT. The result obtained from the INTT undergoes another element-wise multiplication operation with a vector $\boldsymbol{\psi}_{\text{inv}}$, which consists of the multiplicative inverses of the powers of ψ This procedure is fully elucidated in Algorithm 3 and for more information for PWC and NWC techniques reader is referred to Longa and Naehrig (2016).

Algorithm 3 Negative Wrapped Convolution Technique

procedure NWC(a, b, ψ, ψ_{inv})
  $a_{\text{new}} \leftarrow a \odot (1, \psi, \psi^2, \ldots, \psi^{n-1})$
  $b_{\text{new}} \leftarrow b \odot (1, \psi, \psi^2, \ldots, \psi^{n-1})$
  $a_{\text{NTT}} \leftarrow \text{NTT}(a_{\text{new}})$
  $b_{\text{NTT}} \leftarrow \text{NTT}(b_{\text{new}})$
  $c_{\text{NTT}} \leftarrow a_{\text{NTT}} \odot b_{\text{NTT}}$
  $c \leftarrow \text{INTT}(c_{\text{NTT}})$
  $c_{\text{final}} \leftarrow c \odot (1, \psi_{\text{inv}}, \psi_{\text{inv}}^2, \ldots, \psi_{\text{inv}}^{n-1})$
  return c_{final}

These techniques obviate the need for doubling the input polynomials and prevent polynomial reduction after the INTT operation, substantially reducing the computational time. However, NWC entail additional pre-processing and post-processing steps; these involve multiplication operations with $\boldsymbol{\psi}$ and $\boldsymbol{\psi}_{\text{inv}}$ vectors.

Given that the formulas in Eqns 2.1 and 2.2 lead to a quadratic complexity of $O(n^2)$, there are various algorithms available to perform NTT operations more efficiently, including the iterative NTT, merged NTT, and four-step NTT algorithms. These algorithms utilize reductions over cyclotomic polynomials to enhance efficiency. Specifically, these algorithms operate in the ring $\mathbb{Z}_q[x]/\Phi_m(x)$, working with different bases — some with x^n-1 and others with x^n+1 — facilitating more effective computations in certain contexts.

A series of algorithms incorporating butterfly operations have been developed to enhance the efficiency of NTT and INTT operations. These operations significantly reduce time complexity in polynomial computations, offering faster and less complex solutions.

The Cooley-Tukey (CT) butterfly operations and the Gentleman-Sande (GS) butterfly operations, both emphasized by Chu and George (1999), are crucial components in NTT algorithms. The CT butterfly operations are generally utilized during the NTT stage, facilitating the merging of polynomials while accelerating the processes and reducing complexity. This operation employs three inputs, named u, t, and w (w is in fact a twiddle factor), to produce two outputs: $(u+tw) \bmod q$ and $(u-tw) \bmod q$. Modular addition, subtraction, and multiplication operations are performed during this operation. CT operations are illustrated in Steps 6-9 of Algorithm 4.

On the other hand, the Gentleman-Sande (GS) butterfly operations are mainly preferred in INTT operations, aiding in optimizing processes and assisting in the division of polynomials. In this operation, the modular multiplication of t and w is executed following the modular subtraction, resulting in two outputs: $(u-t)w \bmod q$ and $(u+t) \bmod q$. Both butterfly operations allow for the faster and more efficient execution of NTT algorithms.

The Iterative NTT algorithm illustrated in Algorithm 4 is designed to transform a polynomial into the NTT domain using the Cooley-Tukey butterfly structure. The input polynomial $A(x)$ is received in natural order, while the output is returned in a bit-reversed order. The three nested loops carry out transformations on the polynomial. Each "butterfly" operation generates two new values using the modular arithmetic operations applied on the values defined as u and t. During these operations, a value referred to as w, which is one of the powers of the primitive root of unity, is also used as a multiplier. The algorithm can maintain high efficiency even when operating on large

Algorithm 4 Iterative NTT (Zhang et al. (2020))

Input: $A(x) \in R_{q,n}$ in natural order
Input: Primitive Nth root of unity $w_N \in \mathbb{Z}_p$
Input: A prime number q
Output: $A(x) \in R_{q,n}$ in bit-reversed order

for s from $\log_2(N)$ down to 1 **do**
  $m \leftarrow 2^s$
  for k from 0 to $N/m-1$ **do**
    $w \leftarrow w_N^{\text{bit reverse}(k,\log_2(N)-s)\cdot(m/2)}$
    for j from 0 to $m/2-1$ **do**
      $u \leftarrow A[k\cdot m+j]$
      $t \leftarrow A[k\cdot m+j+m/2]$
      $A[k\cdot m+j] \leftarrow (u+t\cdot w) \mod q$
      $A[k\cdot m+j+m/2] \leftarrow (u-t\cdot w) \mod q$
return A

data sets, as there are many butterfly operations that can be performed separately and independently. This indicates that the algorithm possesses a highly parallelizable structure. This feature significantly reduces the time complexity for large-scale computations, thereby facilitating faster operations.

2.2.0.1 Montgomery Reduction in NTT

In the context of NTT-based polynomial multiplication, one of the most significant challenges is the efficiency of arithmetic operations. Notably, the reduction process in modular arithmetic is among the most time-consuming steps within NTT especially in the modular multiplication operation. The Montgomery reduction is a method developed to address this challenge and accelerate modular reduction operations. By employing arithmetical operations that are less costly compared to the standard modular reduction, this method aims to achieve faster and more efficient results in polynomial multiplication. The specifics of how the Montgomery reduction is applied to enhance the performance of NTT-based polynomial multiplication will be elucidated in this section.

The Montgomery reduction, introduced in Montgomery (1985), is presented in Algorithm 5. This method is renowned for its ability to perform modular reductions without direct divisions, making it attractive for hardware and software implementations, where division is costly. The algorithm takes three inputs: typically c, representing an integer multiplication result; an ℓ-bit modulus q; and $\mu = -q^{-1} \bmod R$. In the first step, the

Algorithm 5 Montgomery Reduction

Input: c (2ℓ-bit positive integer)
Input: Modulus q (l-bit positive integer)
Input: $\mu = -q^{-1} \mod R$ where $R = 2^{\ell} \mod q$
Output: $c = c \cdot R^{-1} \mod q$

1: $u_1 \leftarrow c \cdot \mu \mod R$
2: $u_2 \leftarrow c + u_1 \cdot q$
3: $u_3 \leftarrow u_2 >> \ell$
4: $u_4 \leftarrow u_3 - q$
5: **if** $u_4 < 0$ **then**
6: $\quad c \leftarrow u_3$
7: **else**
8: $\quad c \leftarrow u_4$
9: **return** c

lower ℓ-bits of of c is multiplied by μ and the result is reduced $\mod R$ and stored in u_1. Next, the result of the multiplication of u_1 and q are added to c, which produces a 2ℓ-bit number whose least significant ℓ-bits are all zero. In the subsequent step, the result of the previous step, u_2 is right-shifted by ℓ bits. This operation simulates an integer division by R without executing an actual division. The modulus, q is subtracted from the resulting value u_3 and the result of the subtraction is stored in u_4. As u_4 can be a negative number, the remaining steps ensure that always a positive integer in the correct range is returned.

The algorithm in fact returns $cR^{-1} \mod q$, instead of $cR^{-1} \mod q$, where it is said to be in the Montgomery domain. Therefore, the Montgomery algorithm requires to obtain $c \mod q$ eventually. However, if the modular multiplication involves a constant multiplier, b, in the computation of $ab \mod q$, then $bR \mod q$ is precomputed and stored. Then, the input of Algorithm 5 becomes $cR \mod q$ when a is multiplied by $bR \mod q$ prior to Algorithm 5. Consequently, Algorithm 5 returns $ab \mod q$. As in all modular multiplication operation in the butterfly operations, one operand is always a twiddle factor, which is a constant value, the precomputation technique eliminates the transformation from the Montgomery domain.

Algorithm 6 Merged-NTT Algorithm with CT Butterfly

Input: $A(x) \in R_{q,n}$ in natural order
Input: $\Psi_{table} \in \mathbb{Z}_q$
Output: $B(x) \in R_{q,n}$ in bit-reversed order

$N \leftarrow \text{length}(A)$
$B \leftarrow A$
$l \leftarrow \log_2(N)$
for m_{power} from 0 to $l-1$ **do**
    $m \leftarrow 2^{m_{power}}$
    $t \leftarrow \frac{N}{2m}$
    for i from 0 to $m-1$ **do**
        $z_1 \leftarrow 2 \times i \times t$
        $z_2 \leftarrow z_1 + t$
        $\Psi_{pow} \leftarrow \text{BitReverse}(m+i, l)$
        $S \leftarrow \Psi_{table}[\Psi_{pow}]$
        for z from z_1 to $z_2 - 1$ **do**
            $U \leftarrow B[z]$
            $V \leftarrow (B[z+t] \times S) \mod q$
            $B[z] \leftarrow (U + V) \mod q$
            $B[z+t] \leftarrow (U - V) \mod q$
return B

2.2.0.2 Merged NTT

The Merged NTT algorithm is similar to the iterative NTT algorithm. However, when multiplying polynomials in the polynomial ring defined with $\mathbb{Z}_q[x]/(x^n+1)$ using traditional algorithms such as the iterative NTT, it is necessary to perform pre-processing and post-processing operations using the NWC technique defined above. Therefore, the Merged-NTT algorithm emerges as a significant development in the field of polynomial multiplications based on NTT, as described in Algorithm 6. The presented algorithm accelerates polynomial multiplication operation by avoiding time-consuming coefficient rearrangement steps typically associated with traditional NTT algorithms. In the polynomial ring defined by $\mathbb{Z}_q[x]/(x^n+1)$, the algorithm optimizes polynomial multiplication operations by interacting with n elements instead of the traditional $2n$.

This study integrates the unified pre-processing and NTT strategies formulated by Roy et al. (2014). In Merged NTT, instead of utilizing ω as the n-th root of unity, ψ as the $2n$-th root of unity is employed for the polynomial ring $\mathbb{Z}_q[x]/(x^n+1)$. This approach eliminates the necessity for a preprocessing step at the outset. In this implementation, we leveraged a ψ-table, which contains precomputed powers of ψ given in natural order to optimize operations further. The ψ-table is stored in Block RAMs in the Montgomery

Algorithm 7 Merged-INTT Algorithm

Input: $A(x) \in R_{q,n}$ in natural order
Input: $\Psi_{\text{table}} \in \mathbb{Z}_q$
Output: $B(x) \in R_{q,n}$ in bit-reversed order

$N \leftarrow \text{length}(A)$
$B \leftarrow A$
$l \leftarrow \log_2(N)$
$t \leftarrow 1$
$m \leftarrow N$
while $m > 1$ **do**
    $j1 \leftarrow 0$
    $h \leftarrow \frac{m}{2}$
    for i from 0 to $h-1$ **do**
        $j2 \leftarrow j1 + t - 1$
        $\Psi_{\text{pow}} \leftarrow \text{BitReverse}(h+i, l)$
        $S \leftarrow \Psi_{\text{table}}[\Psi_{\text{pow}}]$
        for j from $j1$ to $j2$ **do**
            $U \leftarrow B[j]$
            $V \leftarrow B[j+t]$
            $B[j] \leftarrow (U+V) \mod q$
            $B[j+t] \leftarrow ((U-V) \times S) \mod q$
        $j1 \leftarrow j1 + 2 \times t$
    $t \leftarrow 2 \times t$
    $m \leftarrow \frac{m}{2}$
$N_{\text{inv}} \leftarrow \text{modinv}(N, q)$
for i from 0 to $N-1$ **do**
    $B[i] \leftarrow (B[i] \times N_{\text{inv}}) \mod q$
return B

form, i.e., $\psi \times R \mod q$. Additionally, the Merged NTT adopts the Cooley-Tukey butterfly structure.

We all integrate the unified post-processing and NTT strategies formulated by Pöppelmann et al. in Pöppelmann et al. (2015). To avoid the reduction step at the end of the iterative INTT, it is necessary to multiply the output polynomial with the powers of ψ^{-1}, a procedure known as the post-processing operation. In this strategy, the GS butterfly structure facilitates this integration; it takes inputs in a bit-reversed order and gives outputs in a normal order. The same authors have previously advocated this method of consolidating the post-processing steps with INTT, thereby orchestrating a unified inverse NTT operation. The technique is delineated in Algorithm 7, illustrating this integrated approach to inverse NTT with the utilization of the GS butterfly structure.

2.2.0.3 Four-Step NTT

The Four-Step NTT method exploits the execution of small size NTT operations by dividing NTT into smaller fragments. Moreover, it enjoys a substantial advantage in parallelization compared to the previously introduced iterative NTT and merge NTT algorithms, thanks to the ability to perform numerous independent NTT operations at a reduced size. Operating with a lower NTT sizes considerably simplifies the reading from and writing to memory for inputs and outputs, making it an ideal choice for efficient solutions. This approach enhances the optimization of the algorithm, allowing for fast and reliable results Dai and Sunar (2015).

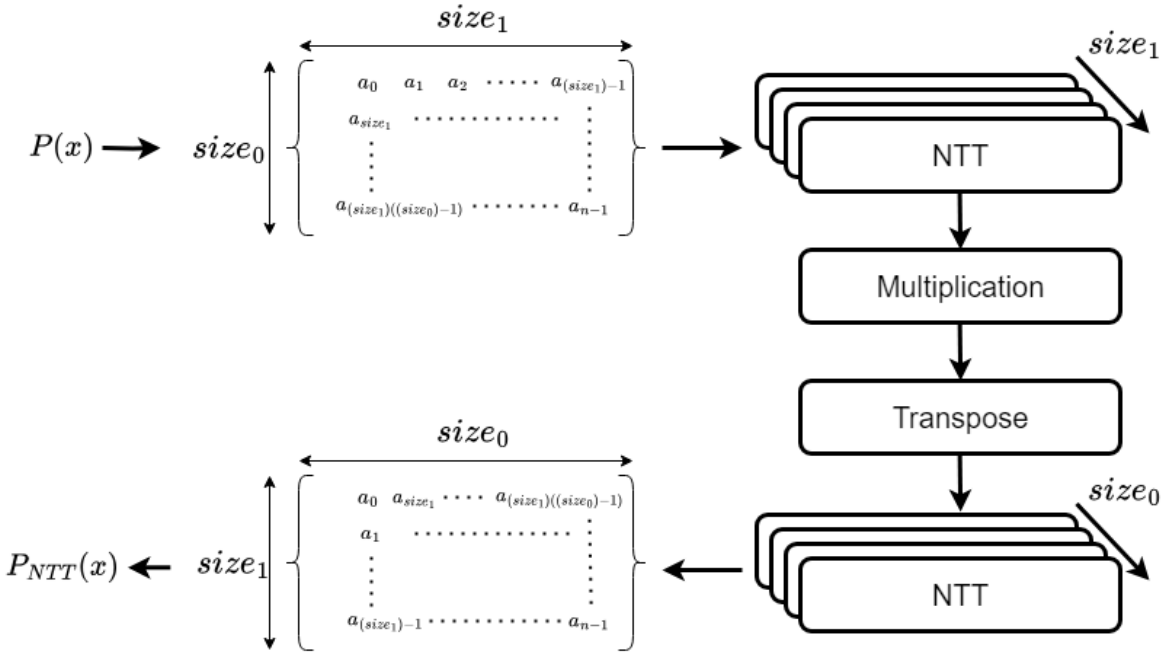

Figure 2.1 This figure illustrates the working principle of the four-step NTT. As can be seen, the process is divided into four main stages: (1) Forward iterative NTT computation; (2) Point-wise multiplication with twiddle factor; (3) Transpose operation; (4) Forward iterative NTT computation.

As depicted in Figure 2.1, the four-step NTT takes a matrix of $size_0 \times size_1$ as input, whose elements are the coefficients of the input polynomial. The parameters $size_0$ and $size_1$ can be modified according to design choices. The four-step NTT comprises four main stages. The first stage is the NTT stage, where independent NTT operations are conducted for each column of the matrix. In this stage, the size of the NTT operations is determined by $size_0$, while they are performed up to $size_1$ times. The second stage is the multiplication stage, in which each polynomial coefficient is point-wise multiplied by a corresponding power of the twiddle factor. The third stage involves transpose operation,

Algorithm 8 Four-step INTT using GS-based butterfly structure

Input: $A_{NTT}(x) \in R_{q,n}$ in bit-reversed order
Input: Primitive Nth root of unity $w_N \in \mathbb{Z}_p$
Input: Matrix sizes $size = [size_0, size_1]$
Output: $A(x) \in R_{q,n}$ in natural order

Convert the polynomial A_{NTT} into a 2D matrix, $Matrix0_A$ with dimensions $[size_1][size_0]$
for i from 0 to $size_1 - 1$ **do**
    for j from 0 to $size_0 - 1$ **do**
        $Matrix0_A[i]$.append($A_{NTT}[((j \cdot size0) + i)\%n]$)
$W_0 \leftarrow W^{n/size_0} \mod q$
$W0_{table} \leftarrow$ generateTwiddleFactorTable($W_0, size_0, q$)
for i from 0 to $size_1 - 1$ **do**
    $Matrix0_A[i] \leftarrow$ GS-INTT($Matrix0_A[i], W0_{table}, q$)
$W_{table} \leftarrow$ generateTwiddleFactorMainTable($size, q, W$)
Perform point-wise multiplication with W_{table} to get $Matrix1_A$
$W1 \leftarrow W^{n/size_1} \mod q$
$W1_{table} \leftarrow$ generateTwiddleFactorTable($W_1, size_1, q$)
for i from 0 to $size_0 - 1$ **do**
    $Matrix1_A[i] \leftarrow$ GS-INTT($Matrix1_A[i], W1_{table}, q$)
Perform Transpose operation to get $Matrix2_A$
Convert $Matrix2_A$ from 2D array to 1D array A
$N_{inv} \leftarrow$ modinv(n, q)
for i from 0 to $n - 1$ **do**
    $A[i] \leftarrow (A[i] \cdot N_{inv})\%q$
return A

wherein a transpose operation is applied to the matrix after the NTT operation. The fourth and final stage is another NTT stage, where the NTT operation is applied to each column of the transposed matrix. This four-stage process holds critical importance for conducting polynomial operations swiftly and efficiently.

In the initial NTT stage of the Four-Step NTT, a Cooley-Tukey-based NTT is employed. Given the NTT inputs in the normal order, this approach yields outputs in the bit-reversed order. During the multiplication stage, the multiplication of twiddle factors is also organized according to the bit-reversed order. Following this, a transpose operation is applied. This step ensures the columns of the matrix are sorted according to the bit-reversed order while each element in a column retains a normal order arrangement. Finally, in the final stage of the Four-Step NTT, a Cooley-Tukey-based NTT is utilized again. This strategy guarantees that the resultant output maintains a bit-reversed order arrangement akin to the outcome in the Merge NTT.

The four-step INTT comprises the four stages introduced above and is illustrated in Algorithm 8. A distinctive feature of INTT, as opposed to NTT, is that it employs a GS-based butterfly structure. This version of the INTT algorithm is favored because it takes inputs in bit-reversed order and gives outputs in normal order. Additionally, unlike in four-step NTT, the polynomial resulting from the final stage of the four-step INTT algorithm is multiplied by n^{-1} in $\mathbb{Z}_q$. Functions $generateTwiddleFactorMainTable$ and $generateTwiddleFactorTable$ are tasked with generating arrays to be used in the stages of GS-INTT and twiddle factor multiplication, respectively, based on the given twiddle factor.

In the traditional NTT approach, processing extremely high-degree polynomials becomes challenging due to memory insufficiencies, posing a significant limitation, especially in large-scale computations. The Four-step NTT method addresses this issue by partitioning high-degree polynomials into smaller vector segments. This partitioning process facilitates the breakdown of polynomials into smaller, more manageable pieces, allowing each component to be processed individually. Moreover, this approach enables the utilization of off-chip memory, thus ensuring that even high-degree polynomials can be effectively handled. Coupled with an appropriate data pathway, this strategy paves the way for more efficient polynomial operations. Consequently, the Four-step NTT can substantially reduce the memory requirements for high-performance computations.

2.3 Homomorphic Encryption

HE refers to a cryptographic method that facilitates the execution of operations directly on encrypted data without the necessity of decryption first. Essentially, this encryption not only protects the data but allows for secure computations to occur, resulting in an outcome that, when finally decrypted, identical precisely to what would have been achieved if the calculations were carried out on the raw, unencrypted data. The cornerstone of this encryption approach is its capacity to retain the confidentiality of the data all through the computational stages, thereby ensuring an uncompromised privacy-safeguarding avenue while allowing for necessary data manipulations and analyses.

Figures 2.2 and 2.3 respectively illustrate the traditional and HE methods in the context of third-party computation. In the traditional encryption method depicted in Figure 2.2, data encrypted by the user is sent to a third party for processing. At this stage, the

third-party decrypts the data to perform the necessary operations and then re-encrypts the results before sending them back to the user. This process exposes the data at one point, creating security risks.

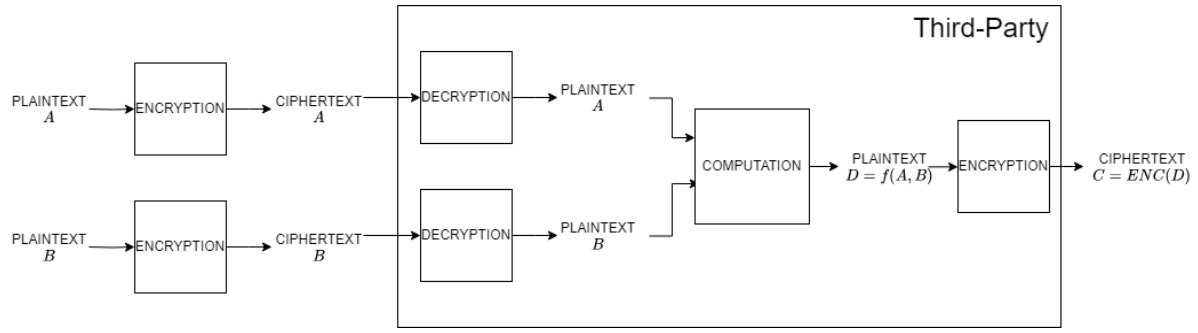

Figure 2.2 The workflow of the traditional encryption method incorporating third-party computation, delineating each critical step in the process.

On the other hand, the HE method is detailed in Figure 2.3. This method describes a process, where the data encrypted by the user can be directly processed without decryption. In other words, the third party can perform operations on the encrypted data and send the results back to the user, ensuring continuous (or end-to-end) protection of data confidentiality. This comparison underscores the persistent confidentiality afforded by the HE method against the potential security risks of traditional encryption. While HE offers a higher level of security by eliminating the necessity of decryption, the traditional method, albeit providing a simpler workflow, engenders a security vulnerability during operations.

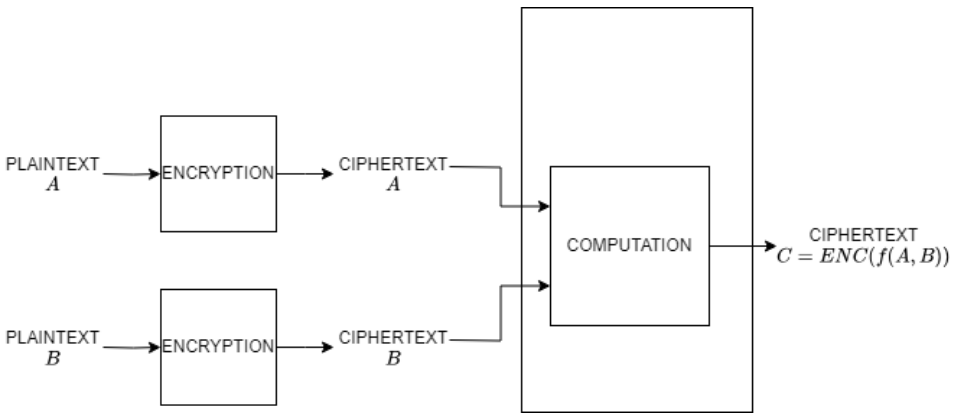

Figure 2.3 Illustrating the process of third-party computation facilitated through homomorphic encryption, outlining each pivotal phase.

As we explore the details of HE, it is essential to grasp the idea of "noise". This key feature helps differentiate between different types of HE. At its core, "noise" means the extra random data added, which is sampled from a random distribution (e.g., Gaussian

with zero mean and small standard deviation) during encryption. This additional data helps keep the information secure, but it can also make calculations on the encrypted data more complex. As Acar et al. (2018) have outlined, we can divide HE into three main types; namely partial homomorphic, somewhat homomorphic, and fully homomorphic encryption, each distinguished based on how the "noise" is handled.

- **Partial Homomorphic Encryption (PHE):** PHE is an encryption scheme that supports a limited set of operations. In the PHE scheme, either addition or multiplication operations can be performed, but it cannot support both operations simultaneously. This characteristic makes application simpler than Somewhat and Fully HE schemes. However, this allows only specific types of operations to be conducted on the encrypted data, limiting its application range. Partial homomorphic encryption schemes are instantiated using conventional algorithms based on number theoretic approach such as Paillier encrytpion scheme Paillier (1999), which does not use noise.
- **Somewhat Homomorphic Encryption (SWHE):** SWHE allows for a broader set of operations to be performed on encrypted data, encompassing addition and multiplication operations. However, this comes with a limitation: after a certain number of operations, the "noise" in the encrypted data grows to a point where it can no longer be decrypted correctly; this necessitates the conscious management of noise growth to prevent decryption errors.
- **Fully Homomorphic Encryption (FHE):** FHE represents HE schemes that allow an unlimited number of both addition and multiplication operations to be performed on encrypted data, without decrypting it. The revolutionary concept that makes this method possible is a technique called "bootstrapping", developed by Gentry, which effectively reduces the noise level, enabling even extensive calculations to be decrypted correctly. While this technique is computationally intensive, it paves the way for securely performing complex operations on encrypted data, opening avenues for secure multi-party computations and privacy-preserving data analyses.

In this thesis, we elect to focus on SWHE. The fundamental reason behind this choice is that this encryption offers a balance between partial and fully homomorphic encryption methods. SWHE supports a broader set of operations compared to PHE, allowing for more complex computations and offering a more comprehensive range of applications. On the other hand, there is a performance overhead in FHE due to the bootstrapping operation utilized. Although SWHE requires conscious noise management, it does not

necessitate the application of advanced, overly complicated, techniques such as bootstrapping. With these features, SWHE provides advantages regarding the applicability and efficiency of HE algorithms compared to FHE, making it a more favorable choice in this context.

In SWHE, there are two commonly used schemes in academia as well as industry. The first one is the BFV scheme introduced in Brakerski et al. (2011). This scheme supports operations over integers and allows both addition and multiplication operations. The main advantage of the BFV scheme is that the results of the computations are entirely accurate, which is crucial for secure and error-free calculations. However, this scheme is primarily optimized for integer calculations and does not support operations involving real numbers.

The second is the CKKS scheme, initially introduced by Cheon, Kim, Kim, and Song in Cheon (2017). This scheme allows for the approximate addition and multiplication of encrypted messages. CKKS supports integers and accommodates complex and real numbers, granting it greater flexibility for numerous applications. The attributes of the CKKS scheme are particularly vital in applications such as machine learning or statistical analysis because these fields often require working with data that involve integers and real or complex numbers. Moreover, the ability of the CKKS scheme to perform approximate calculations makes it a suitable solution for a wide range of applications.

2.3.1 Cheon-Kim-Kim-Son Homomorphic Encryption Scheme

The CKKS scheme, like other HE schemes, is based on the ring-LWE (RLWE) problem (Lyubashevsky et al. (2013)). This method adopts a decryption structure of the form $(m + e \bmod q)$, where m represents the original message, e is a small error term added to ensure security, and q represents the encryption modulus. The noise added during the encryption process can replace the original message in approximate arithmetic operations, preserving the significant digits of the main message.

In the CKKS scheme, the "rescaling" technique is developed, leveraging the scaling factor "p" to manipulate the modulus of the encrypted messages. In this technique, p is chosen as a scaling factor and is utilized during the encryption process to perform manipulations on the cipher text. That is, starting with a cipher that has an encryption $\langle c, \mathrm{sk}\rangle \equiv m + e \pmod q$, a new cipher is obtained using the scaling factor p as $\lfloor p^{-1} \cdot c \rceil \pmod{q/p}$, which is a valid encryption of the message m/p with an approximate error of e/p. This procedure

not only diminishes the size of the ciphertext modulus but also reduces the error term, thereby preserving the precision of the encrypted texts.

This methodology results in a precision loss of only one more bit compared to unencrypted floating-point arithmetic, demonstrating its suitability for approximate computations. One of the most significant advantages of this approach is that the required bit size of the ciphertext modulus grows linearly with the depth of the evaluated circuit, eliminating the necessity for exponential growth.

In the discussions above, a perspective has been provided on the general features of the CKKS scheme and its advantages in practice. However, to fully grasp how this scheme operates in practice, it is essential to carefully examine its fundamental operations, particularly key generation, encryption, and decryption. Below, the critical operations of the CKKS scheme are detailed.

- **Key Generation**:
  - A secret key $sk \in \mathcal{R}_q = \mathbb{Z}_q/\Phi(x)$ is generatee by sampling its coefficients from the set $\{-1, 0, 1\}$ using a uniform distribution. Keeping this key confidential is essential.
  - Then, the public key $(a, b) \in \mathcal{R}_q^2$ is generated as follows
    - $a \in \mathcal{R}_q$ is generated by sampling its coefficients from a uniform distribution.
    - An error polynomial $e \in \mathcal{R}_q$ is generated by sampling its coefficients from a normal distribution with zero mean and small standad deviation (e.g., $\sigma = 3.6$).
    - $b \leftarrow -a \cdot sk + e \bmod q$ is computed.
  - The public key is denoted as $pk = (b, a)$.
- **Encryption**:
  - For a given message $m \in \mathcal{R}_q$, encryption is done using the public key $pk = (pk_0, pk_1)$ and the resulting ciphertext is $ct = (c_0, c_1)$.
  - $\mu \in \mathcal{R}_q$ is generated by sampling its coefficients from a uniform distribution.
  - The error polynomials e_0 and e_1 are sampled the same way e is sampled in the key generation.
  - $c_0 \leftarrow \mu \cdot pk_0 + m + e_0 \bmod q$ and $c_1 \leftarrow \mu \cdot pk_1 + e_1 \bmod q$ are computed.

- **Decryption**:
  - The ciphertext $ct = (c_0, c_1)$ is decrypted using the secret key sk.
  - $m' \leftarrow c_0 + c_1 \cdot sk$ is computed
  - The decryption yields $m' \approx m + e'$, which is the sum of the original message m and a noise e'.

We can show that the decryption works as follows:

$$\begin{aligned} c_0 + c_1 \cdot sk &= \mu \cdot pk_0 + m + e_0 + (\mu \cdot pk_1 + e_1) \cdot sk \\ &= \mu \cdot (-a \cdot sk + e) + m + e_0 + \mu \cdot a \cdot sk + e_1 \cdot sk \\ &= \mu \cdot e + m + e_0 + e_1 \cdot sk \\ &= m + e', \end{aligned}$$

where $e' = \mu \cdot e + e_0 + e_1 \cdot sk$ is a polynomial in $\mathcal{R}$ with small coefficients. As the error polynomial consists of small coefficients, m' approximates the message m.

Microsoft's SEAL library< facilitates homomorphic applications by providing efficient implementations of both the BFV and CKKS schemes. This library employs the full RNS variant of both the CKKS and BFV schemes, and it partitions high-precision polynomial arithmetic into smaller low-precision integer arithmetic, allowing operations to be executed in parallel while eliminating division and rounding processes. Thanks to these features, the BFV and CKKS schemes offer suitable solutions for a wide range of applications by providing faster and more efficient homomorphic computation.

2.3.2 Residue Number System

The Residue Number System (RNS) plays a pivotal role in many cryptographic applications because it facilitates parallel arithmetic computations for large numbers. This is achieved by utilizing a series of smaller moduli instead of a single large modulus, hence offering the opportunity to employ single-precision arithmetic instead of the more complex and computationally intensive multi-precision arithmetic. As a result, more efficient HE implementations (e.g., Türkoğlu et al. (2022) which presents an efficient implementation of BFV homomorphic encryption scheme) can be developed.

Additionally, RNS arithmetic exhibits significant efficiency in considerably enhancing the

processing speed of R-LWE-based lattice SWHE schemes. Furthermore, efforts have been initiated to devise RNS variants of such encryption schemes, which have demonstrated substantial speed improvements on platforms fully leveraging the parallelism afforded by RNS, including GPU and FPGA environments.

By the Chinese Remainder Theorem (CRT) (Pei et al. (1996)), we consider an integer X that is less than a predefined M. This X can be represented using a series of residues, denoted as x_i, derived through the equation $x_i = X \bmod m_i$ across an index range from 1 to r, where m_i are pairwise relatively prime moduli. This representation stands valid under the condition that M equals the product of m_i for all i in the mentioned range. Furthermore, every m_i belongs to a group characterized by pair-wise relatively prime numbers, an ensemble generally termed as the moduli. Consequently, a conventional notation adopted is $[X]_{m_i} = X \bmod m_i$. Leveraging this foundational concept, we can employ CRT to craft mathematical expressions grounded in this residue representation.

$$|X|_M = \left| \sum_{i=1}^{r} \left| x_i \cdot M_i^{-1} \right|_{m_i} \cdot M_i \right|_M, \quad \text{for } i = 0, 1, \ldots, r-1, \tag{2.4}$$

$$\text{where} \quad M_i = \frac{M}{m_i}.$$

In Equation 2.4, the primary objective is to reconstruct the original integer X utilizing the provided residues x_i. Here, each x_i represents the residue of X with respect to a specific modulus m_i. The term M_i^{-1} denotes the modular inverse with respect to m_i and ascertains the weighting used during the reconstruction process. M_i is an integer by obtaining by multiplying all moduli except for m_i and determines the positional significance of each residue within the RNS structure. Consequently, the summation in the equation provides a weighted sum of these residues, facilitating the reconstruction of X.

2.4 Field Programmable Gate Array (FPGA)

Field-Programmable Gate Arrays (FPGA) hold a significant place in the electronics realm. Comprising programmable logic blocks and the interconnection paths that facilitate information flow between them, FPGA represents a type of digital integrated circuit. This unique structure grants it extensive applicability across various domains,

leading to its preference in industrial and academic projects. Remarkably, the ability to program the hardware flexibly and its adaptability stand as primary reasons behind FPGA's widespread popularity.

The components within an FPGA are paramount to its functionality and versatility. Each element has a distinct role and contributes to the overall efficiency and capability of the device. Logic blocks handle computational tasks, programmable interconnects provide the communication backbone, and various memory elements store data, all under the precise control of the designer. The importance of these components cannot be overstated, as they collectively determine the FPGA's potential in complexity and performance.

- **Look-Up Tables (LUTs):** LUTs are fundamental functional components of the FPGA. They possess the capacity to perform a variety of arithmetic and logical operations, handling tasks ranging from simple processes to complex computations.
- **Programmable Interconnects:** These components interlink different sections of the FPGA, ensuring uninterrupted and efficient data and signal flow.
- **Input/Output Blocks:** These blocks enable the FPGA to send and receive signals to and from other devices.
- **Memory Elements:** FPGAs are equipped with various memory elements to cater to data storage needs. Depending on the specific requirements of the FPGA design, different types of memories can be utilized.
  - **Flip-Flops:** These are the most basic memory elements used in FPGAs and can store a single bit of data, either a 0 or a 1. They are commonly utilized within logic blocks.
  - **Block RAM (BRAM):** Compact memory segments embedded within the FPGA. They have a larger storage capacity than Flip-Flops and are typically used for data buffering and other purposes.
- **DSP Slices:** Highly optimized hardware units designed to specific mathematical operations such as multiplications.

Implementing HE operations and NTT with large ring sizes on FPGA introduces several challenges. The limited memory capacity of FPGAs complicates managing extensive data volume. The substantial data volume required by HE and NTT operations accentuates these limitations. Data movement and management become involved for such applications. Specifically, ensuring efficient data reading and writing patterns with BRAM blocks, which we use to store polynomial coefficients for recursive algorithms like NTT, is

notably challenging. This highlights the importance of efficient data movement to achieve optimal performance. On another note, modular multiplication, a pivotal operation in HE, is complex and demands efficient implementation on hardware. This operation can lead to substantial resource consumption and extended computation times.

Optimizing the hardened IP functions embedded in FPGA demands a thorough understanding of both the hardware specifics and the intricacies of the application. For instance, our adaptation of the word-level Montgomery reduction method, utilized to maximize the potential of DSP blocks, reflects these optimization challenges. The FPGA's limited hardware resources, especially logic blocks and memory, necessitate efficient design to utilize these assets. These constraints become prominent when considering the parallelization requirement of the CKKS operations; also, the finite availability of DSP blocks restricts this potential. Furthermore, the reprogrammable nature of FPGAs can complicate adapting existing designs for different parameters and ring sizes. Nonetheless, despite these hurdles, a successful implementation of NTT on FPGA is attainable with the right design and optimization strategies.

3. OUR WORK

This chapter delves deeply into the methodologies we employed to realize our architectures. Initially, we introduce our approach to designing NTT-based polynomial multiplication architectures with temporal flexibility. We present the unified butterfly unit amalgamating modular arithmetic operations in this context. Our research extends to implementing the Parametric Merged NTT Hardware, which caters to specific optimization needs and its four-step variant.

As the chapter progresses, the focus narrows to HE, especially the CKKS scheme. We delve into our specially designed hardware architecture with temporal flexibility to accelerate the operations of the CKKS scheme. Detailed discussions are presented about the full RNS variant of the CKKS scheme and the intricacies of the homomorphic multiplication and key-switching mechanisms. The concluding sections showcase our unified architectural design developed for homomorphic multiplication and key-switching, demonstrating its efficiency and adaptability.

3.1 Efficient Design-Time Flexible NTT-Based Polynomial Multiplication

Architectures

This section will focus on two distinct hardware architectures we developed for NTT-based polynomial multiplication, which is a follow up study of the work presented in Mert (2021). Firstly, we will delve deeply into the unified butterfly unit architecture

concept, which plays a critical role in performing modular operations and forms the foundation of our approach. This concept expands and offers a detailed examination of the ideas proposed initially in Mert (2021), aiming to present a comprehensive analysis that underscores its significance and functionality. Subsequently, we will thoroughly explore our designs, specifically the Design-Time Flexible Merged NTT and the Four-Step NTT architectures. This section will discuss our expectations from both architectures and their potential advantages and limitations. We will scrutinize these architectures' technical specifications, performance metrics, and application domains in depth.

3.1.1 Unified Butterfly Unit

The Unified Butterfly Unit, designed to perform the modular arithmetic operations required for homomorphic operations, such as those for executing NTT, INTT, modular addition, subtraction, and multiplication tasks. Aiming for a high-performance hardware architecture, we use the merged NTT and four-step NTT algorithms introduced above. In these algorithms, the combined butterfly unit is employed to carry out both CT and GS butterfly operations, a unit which was analysed in detail in Mert (2021).

The unified butterfly unit is adept at conducting both CT and GS butterfly operations, while simultaneously handling modular addition, subtraction, and multiplication. This unit processes three 32-bit coefficients (a, b, w) and two control signals (ct, mt) as inputs. In return, it produces five 32-bit coefficients (e, o, s, $m0$, $m1$) as outputs. The configuration of the control signals ct and mt determines the operation. Specifically, in the configuration $ct, mt = (0,0)$, the outputs e and o represent $a+b \mod q$ and $(a-b)\cdot w \mod q$, respectively. On the other hand, for the configuration $ct, mt = (1,0)$, the outputs e and o stand for $a+b\cdot w \mod q$ and $a-b\cdot w \mod q$. These outputs from the two configurations, e and o, correspond to the CT and GS butterfly operations discussed in Section 2.2. The other outputs are utilized for additional modular operations, such as addition, subtraction, and multiplication. Specifically, for the configuration $ct, mt = (0,1)$, the output $m0$ is used for the operation $(a+b)w \mod q$, while the output $m1$ is used for the operation $(a+b)w \mod 2^{32}$.

3.1.1.1 Modular Addition and Subtraction Units

Modular addition and subtraction operations play a critical role in the unified butterfly structure. Efficiently executing these operations has a significant impact on the overall performance. A hardware module designed for the modular addition operation is visualized in Figure 3.1.

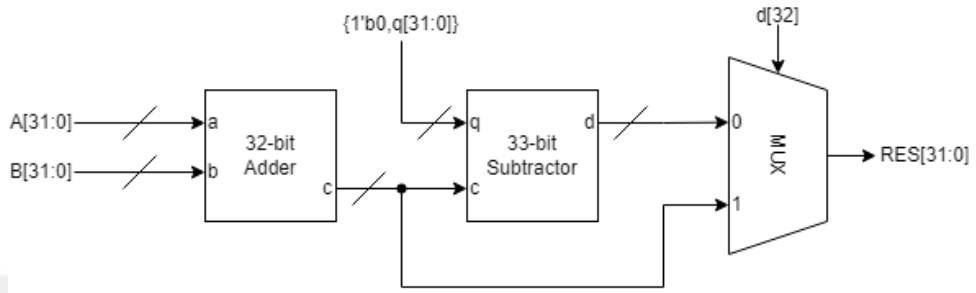

Figure 3.1 32-bit Modular Adder Unit

This module takes two 32-bit numbers (A and B) and a modulus value (q) as inputs. These two numbers are added, and the modular value of the sum concerning q is directed to the output. If the result exceeds the modulus value, it is subtracted from this value to attain the actual modular result. In addition to the traditional addition operation, an additional comparison and conditional assignment are performed during the modular addition process. This ensures that the resulting sum is checked against the modulus value, and the appropriate value is assigned to the output based on the outcome.

The modular subtraction operation is structurally similar to the modular addition module, but the primary function is subtraction rather than addition. A subtraction operation is performed for two inputs. If the subtracted result is negative, a correction step is required. This is achieved by adding the result to the modulus value. After the correction process, the resulting value is checked to ensure it is either positive or zero. If the value is positive or zero, it is directly assigned to the output. However, the corrected value is used if a negative result is obtained. This method ensures that the results obtained in the modular subtraction process always fall within a specific positive range.

3.1.1.2 Modular Multiplication Unit

The Modular multiplication unit is a critical component, especially since it is the most computationally intensive part of the NTT and INTT operations. The word-level Montgomery modular multiplication method is adopted in the unit designed in Mert et al. (2019). This method is chosen to execute the modular reduction step, which is the most compute-intensive section of the unified unit. Mert et al. (2019)'s selection of the "word-level Montgomery reduction" method proves more efficient than other popular methods such as the Barrett reduction (Barrett (1986)),K2-RED (Bisheh-Niasar et al. (2021)), and Plantard(Huang et al. (2022)).

The design-time configurability of the Montgomery reduction algorithm makes this method more efficient. However, the traditional Montgomery algorithm discussed in Section 2.2.0.1 must be more suitable for DSP blocks. Although each DSP block can perform complex operations, the input and output bit values are inappropriate for regular Montgomery reduction. On the contrary, the word-level Montgomery method, as shown in Algorithm 9, divides the reduction process into several steps, allowing us to operate with lower bits. As a result, employing the word-level approach enables us to utilize DSP blocks more effectively and simplifies implementing these operations on the DSP.

In our research, we employ an NTT-friendly prime that enhances the efficiency of the Montgomery reduction algorithm. In the conventional Montgomery method, as presented in Algorithm 5, the first two steps involve intricate multiplication operations. However, thanks to the properties of the NTT-friendly prime, these complex multiplication tasks are transformed into simpler shift operations in the *Word-Level Montgomery* algorithm, as described in Algorithm 9. This transformation is particularly crucial in FPGA implementations because it allows us to convert the computationally intensive multiplication operations into shift operations, which are naturally more performant on hardware platforms such as FPGA.

3.1.2 The Parametric Merged NTT Hardware

Developing an NTT framework on an FPGA platform presents considerable challenges, primarily due to the intricate Memory Access Patterns (MAP) required to attain a high throughput. Integrating a variety of butterfly units not only facilitates more flexible resource distribution, but is also crucial in FPGA environments, where resource availability

Algorithm 9 Word-Level Montgomery Reduction Algorithm for NTT-friendly Primes Mert (2021)

Input: $D = A \cdot B$ (a l-bit positive integer, $w \cdot (L-1) \leq l < w \cdot L$)
Input: w $= \log_2(2n)$ (word size)
Input: L $= l/w$ (repeat count)
Input: q (an l-bit positive integer, $q = q_H \cdot 2^w + 1$)
Input: $\mu = -q^{-1} \bmod R$ where $R = 2^{w \cdot L} \pmod{q}$
Output: $C = D \cdot R^{-1} \pmod{q}$

$T3 \leftarrow D$
for i = 0 to L-1 **do**
    $T_{1,H} \leftarrow T_3 \gg w$ # *Upper part of* T_1
    $T_{1,L} \leftarrow T_3 \pmod{2^w}$ # *Lower part of* T_1
    $T_2 \leftarrow -T_{1,L} \pmod{2^w}$
    carry $= T_2[w-1] \vee T_{1,L}[w-1]$
    $T_3 \leftarrow T_{1,H} + (q_H \cdot T_2) +$ carry
$T_4 \leftarrow T_3 - q$
if $T_4 < 0$ **then**
    $C \leftarrow T_3$
else
    $C \leftarrow T_4$
return C

might be restricted.

The work in Mert (2021) created an open-source and parametric hardware design for merged-NTT, capable of adapting to a broad spectrum of ring dimensions and numbers of butterfly units. Distinct MAPs are necessitated at each of the $\log_2 n$ phases of the NTT, meaning that the memory storage locations for coefficients must be meticulously predetermined at every stage to avoid data clashes.

Our approach leverages a universal MAP applicable to ring dimensions ranging from 2^{10} to 2^{15}. Furthermore, we design and realize a parametric hardware solution adaptable at the design stage to facilitate merged NTT/INTT processes based on specified polynomial degrees and butterfly unit numbers.

In the architecture of the overall design, the *unified butterfly unit*, introduced as a PE, is detailed in Section 3.1.1. This PE is a versatile unit designed to execute both CT and GS operations in a unified fashion and the specific butterfly operation can be selected at runtime. In addition, it can perform other modular arithmetic operations, such as modular multiplication and modular addition/subtraction for 32-bit numbers. The unit's functionality can be configured during runtime, providing enhanced flexibility in operations. An address generator is incorporated to produce BRAM read/write addresses

for read/write operations. The control unit is responsible for adjusting the operating mode of the input multiplexers for the unified butterfly units and the address generator. Multiplexers (Mux) are utilized to sequence the input and output data to ensure proper read/write operations. The BRAM groups store coefficients for input polynomials and twiddle factors.

To implement the NTT algorithm efficiently on hardware, a specialized Python script was developed. This script is tailored to automatically generate the required code in a hardware description language by taking into consideration two crucial parameters:

1.1 The size of the polynomial ring (n)

1.2 The number of PEs, which essentially represents the count of butterfly units in the architecture.

With these parameters provided, the script computes the necessary number of memory units for polynomial storage using the formula

$$\text{Number of memory units} = 2 \times \text{Number of PEs}, \tag{3.1}$$

as each butterfly operation reads two vector elements and outputs two vector elements as well. This significant detail is delineated in Figure 3.2, where each PE necessitates two inputs (coefficients of the input polynomial) per clock cycle. In addition, each PE requires one twiddle factor power, implying that the total number of memory units designated for the twiddle factors is equivalent to the number of PEs. In Figure 3.2, "Data" and "TW" represent the input vector elements and twiddle factors, respectively. Furthermore, the data storage capacity for each memory unit is given by

$$\text{Data storage per memory unit} = \frac{n}{\text{Number of memory units}}. \tag{3.2}$$

This systematic and automated approach ensures accuracy and enhances scalability for varying ring sizes and processing capacities.

In our hardware realization of the NTT algorithm, a pivotal component is the control unit designed for orchestrating read and write operations by producing the correct indices (or their addresses in the BRAMs) for input vector and twiddle factors. This control unit is fundamentally structured from an address generator combined with an expansive multiplexer. Crucially, the instantiation of the address generator and the multiplexer is parameterized based on the specified PE count and the dimension of the polynomial ring.

The primary function of the address generator is to take the initial signals for both NTT

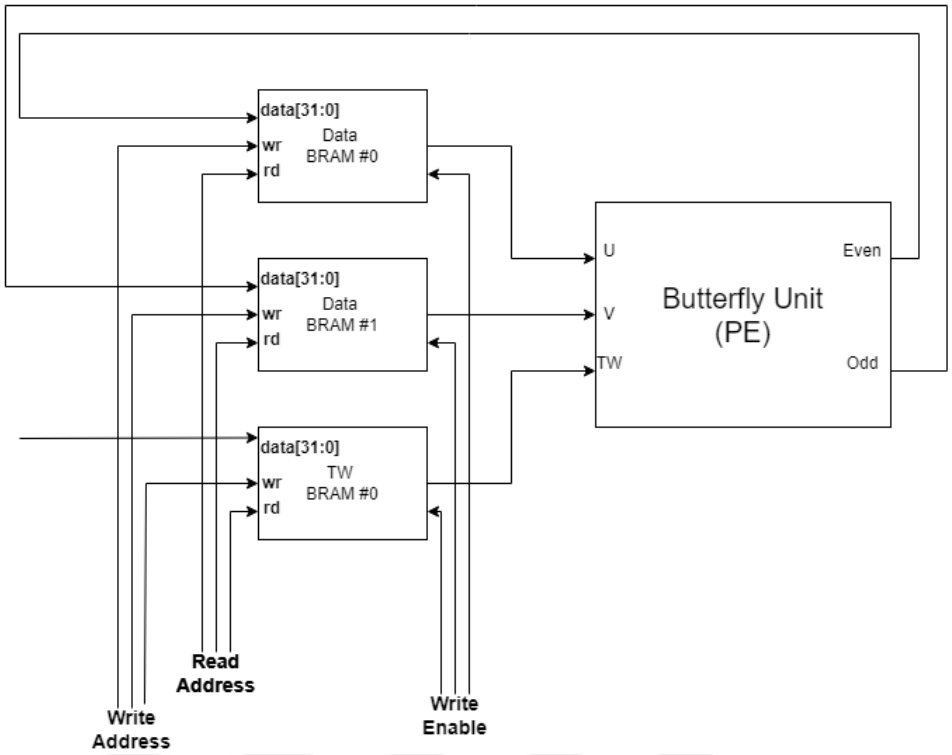

Figure 3.2 BRAM Architecture Overview

and INTT, and consistently generate read and write addresses for the polynomial data with each clock cycle. In tandem with this, it is also entrusted with emitting the requisite read signals that fetch the twiddle factors.

On the other hand, the multiplexer operates by accepting outputs generated by the address unit. It also takes in the even and odd results computed by the butterfly unit during the preceding clock cycle. The multiplexer's responsibility is to transform these outputs into structured arrays. These array-formatted data streams are then channeled as read, write, and data signals towards the BRAM groups. A visual representation of this orchestrated operation can be found in Figure 3.3.

To understand the intricacies of address generation for reading and writing operations, one can consider Figure 3.4 and Figure 3.5. In this example, we have a ring dimension of 8192 and a design conofiguration employing 8 PEs. Initially, our design comprises 16 Block RAMs (BRAMs), each with a capacity to store 1024 addresses

The polynomial's first half (coefficients with indices from 0 to $n/2-1$) is loaded into the even-indexed BRAMs, while the latter is stored in the odd-indexed BRAMs. The reading process initiates from the address 0 (See READ # in Figure 3.4). The subsequent address accessed by each PE is halfway through, precisely at the 256^{th} position (i.e., 512/2). The rationale behind this is to prepare the data for the next stage in the NTT computation.

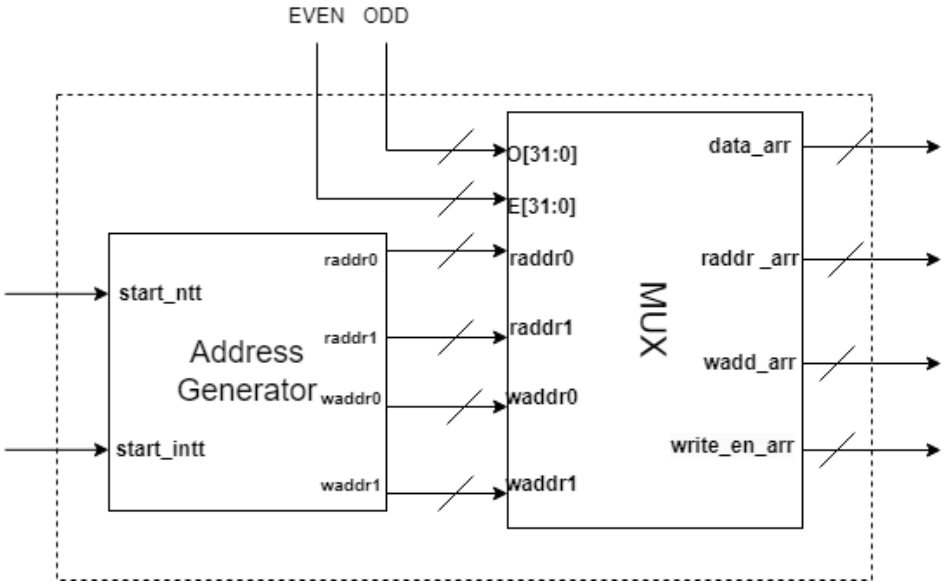

Figure 3.3 Control Unit Architecture Overview

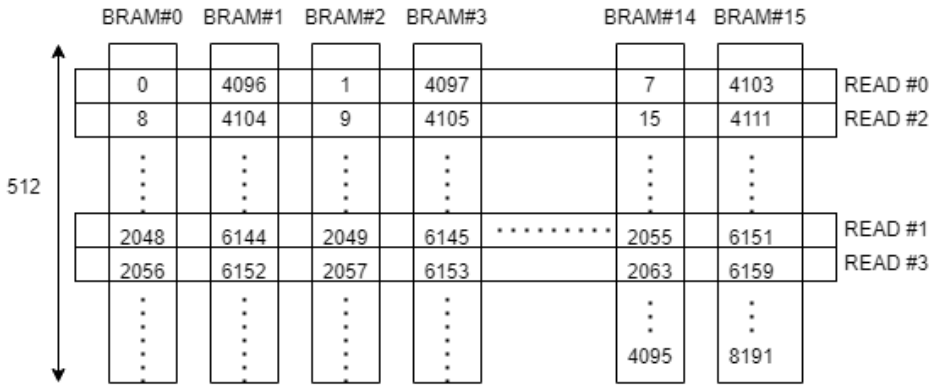

Figure 3.4 Read Addresses of the Memory Access Pattern

For instance, since in the second stage of the NTT computation, the first PE will process the vector elements with indices 0 and 2048, the result of butterfly operation of the vector elements with indices 0 and 4096 in the first stage should be written using the BRAM address 256, where the vector element with index 2048 is currently located. Therefore, it has to be processed in the second clock cycle. It is read from the BRAM in the same clock cycle as the result of the first butterfly operations is written to the same BRAM address. This can be oberserved in Figure 3.5, which shows the memory map of the vector elements in the beginning of the second NTT stage.

Addressing for writing is established between each BRAM pair. Post the initial stage, data from the second half of the even BRAM assembly is transferred to the first half of the odd BRAMs. Conversely, the odd BRAM's first half is written into the even BRAM's

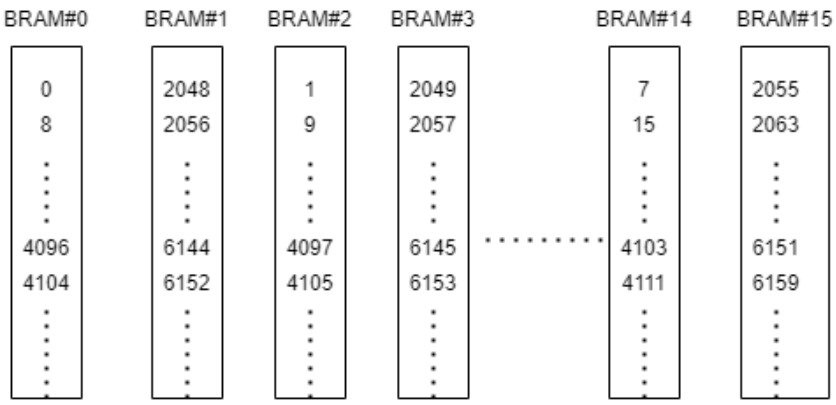

Figure 3.5 Write Addresses of the Memory Access Pattern

latter half. It is worth noting that with every stage progression, both the read and write addresses undergo halving.

In conclusion, through our work on the hardware implementation of the NTT algorithm, we observe that each specific configuration of polynomial size and butterfly unit count necessitates a distinct NTT architecture. Each configuration has its unique MAP and calculation sequence, implying a need for a dynamic structure. To this end, our proposed NTT design proficiently handles this complexity, automatically generating the necessary address and control logic for a specific configuration. This automation approach offers a significant flexibility advantage compared to traditional fixed-configuration applications. Notably, our design's ability to seamlessly adapt to polynomial sizes ranging from 2^{10} to 2^{15} suggests its potential as an effective solution in a wide range of applications.

3.1.3 Four-Step NTT Hardware

The Four-Step NTT algorithm offers distinctive advantages when implemented on hardware platforms, especially in contrast to the Merge NTT approach. This section elucidates these advantages while highlighting the accompanying challenges and trade-offs.

One of the salient advantages of the Four-Step NTT is the ability to achieve parallelization for smaller size NTTs. This stems from the capability to apply the NTT independently across each column of the matrix, into which the input vector to the NTT computation is organized. As a consequence of working with these smaller size NTT operations, memory access patterns are more systematic and organized. This orderly access is pivotal in reducing latency and ensuring efficient data retrieval. Furthermore, the structured memory

access simplifies the establishment of a pipeline architecture, fostering streamlined data flow and potentially boosting processing speed.

While the Four-Step NTT algrithm showcases advantages in terms of parallel processing and memory access, it does necessitate larger memory capacities. The need for a more extensive set of pre-computed twiddle factors directly contributes to increased Block RAM (BRAM) consumption. Careful consideration is required to ensure that this increased memory usage is appropriately managed, especially when deploying the algorithm on constrained hardware platforms.

From the number of operation standpoint, the Four-Step NTT introduces additional multiplications when compared with the Merge NTT. Specifically, while the operation count in the Merge NTT stands at $n/2 \times \log(n)$, the Four-Step NTT algorithm demands additional multiplication operations. These extra operations, although introducing overhead, can be efficiently managed given the other parallelization advantages of the Four-Step approach.

To optimize the throughput of our Four-Step NTT implementation, the First NTT, multiplication, and Second NTT steps are pipelined when deployed on FPGA. We dedicate different PEs for each stage to facilitate this pipelined structure. It is imperative to note that the provided BRAM values and the count of PEs pertain exclusively to one NTT step.

The memory access pattern of the Four-Step NTT stages closely resembles that of the Iterative NTT. The method for determining the number of BRAMs and addresses is consistent with our approach for the Merged NTT. To enhance efficiency in the pipeline structure, we opt for matrix sizes where the number of rows and columns are equal (or close to each other). Our design is configurable and can be easily restructured for different ring dimensions and numbers of PEs. In practice, although our synthesized implementation is based on a ring size of 4096, in the example discussed here, the ring dimension is chosen as 64, the number of PEs is set to 2, and the matrix size is selected as 8×8, for ease of discussion.

In this example, with two PEs, we deploy four BRAMs. This setup aligns with the Merge NTT up to this point. However, a distinct differentiation with the Four Step NTT is in the addressing scheme for the coefficients. Unlike the Merge NTT, where coefficients are placed into even and odd BRAMs consecutively, in our Four Step NTT example, the address progression relies on the row or column count, pivoting on whether it is the first group of NTTs or the second group of NTTs.

In Figure 3.6, the addressing scheme of the presented memory implementation exhibits a

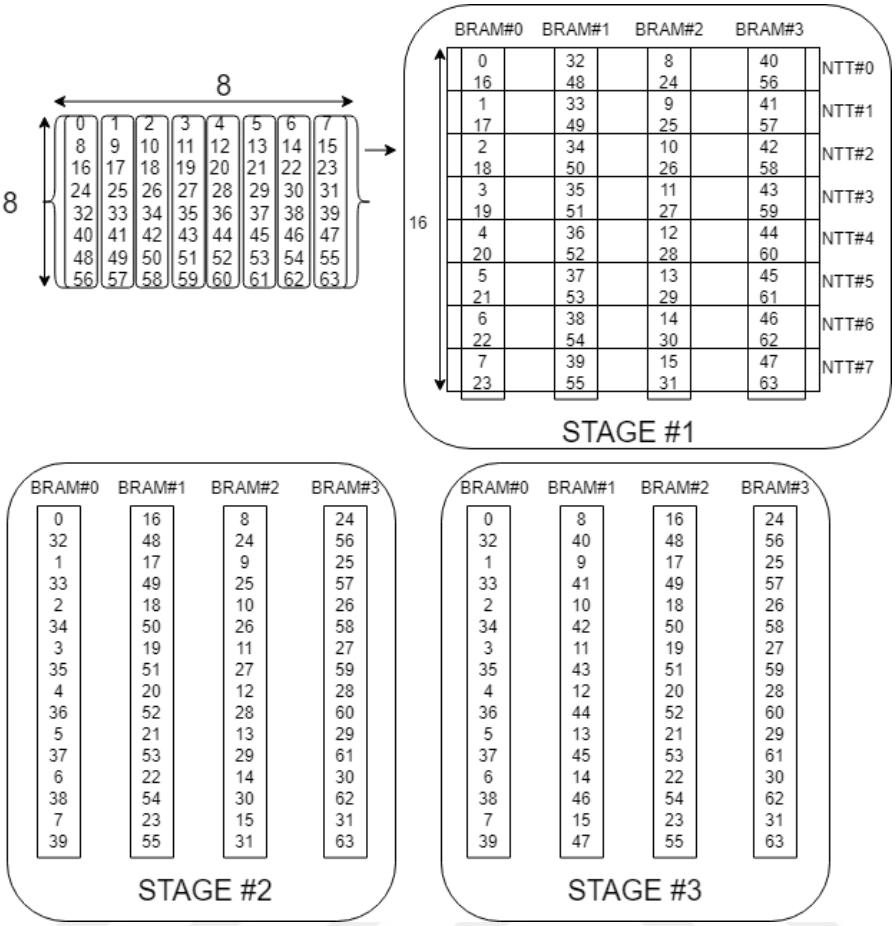

Figure 3.6 Four Step NTT Memory Access Pattern with 2 PEs and $size_0 = size_1 = 8$

distinctive structure from that of the merged NTT. Within this framework, the coefficient at the 0^{th} index is directly allocated to the starting address of the first even-numbered BRAM. However, due to the value of $size_1$ being 8, the coefficient at the 8^{th} index is saved to the first address of the second even-numbered BRAM group. For odd-numbered BRAM addresses, indexing commences from the $size_0/2 \times size_1$ value and increases by $size_1$ (in this case, 8) at each step. This addressing approach continues by incrementing the indexes by one for every $size_0$ element group (e.g., 8).

Following storing the coefficients to BRAMs as indicated above for the NTT algorithm, the address selection for reading and writing operations parallels the approach employed in the Merged NTT implementation. Nonetheless, a salient differentiation in this similarity is the grouping of BRAM address clusters based on the $size_1$ value (e.g., 8). This grouping procedure is represented in the figure through groups labeled with the $NTT\#$ tag. Within each $NTT\#$ group, the selection of read and write addresses is executed as defined in the Merged NTT since each group, in fact, corresponds to an independent, yet much smaller, NTT operation.

The multiplication step in the Four-Step NTT algorithm possesses a simplicity inherent to its point-wise and independent nature. As a result, the memory access pattern is direct and unambiguous. Despite the straightforward nature of the multiplication process, efficiency can be further optimized during the data write-back phase. One notable improvement is integrating the transpose operation during the output write phase to the Block RAMs (BRAMs). This integration eliminates the necessity for a separate transpose step.

The writing strategy employed here uses a formula dependent on the PE and matrix sizes. This formula enables the determination of the precise BRAM address to which each data point will be written. By doing so, memory access is streamlined, and potential bottlenecks are averted. The exact formulas detailing this address calculation are provided in Equations 3.3 and 3.4.

$$newID = \left\lfloor \frac{ADDR \mod \left\lfloor \frac{size_1}{2} \right\rfloor - ADDR \mod \left\lfloor \frac{size_1}{PE} \right\rfloor}{\left\lfloor \frac{size_1}{PE \cdot 2} \right\rfloor + \left\lfloor \frac{ADDR}{\lfloor \frac{N}{PE \cdot 2} \rfloor} \right\rfloor} \right\rfloor \tag{3.3}$$

$$newADDR = size_0 \cdot (ADDR \bmod \left\lfloor \frac{size_1}{PE} \right\rfloor) + \left\lfloor \frac{ADDR \bmod \left\lfloor \frac{N}{PE \cdot 2} \right\rfloor}{\left\lfloor \frac{size_1}{2} \right\rfloor} \right\rfloor + \left\lfloor \frac{size_0}{PE} \right\rfloor \cdot ID \tag{3.4}$$

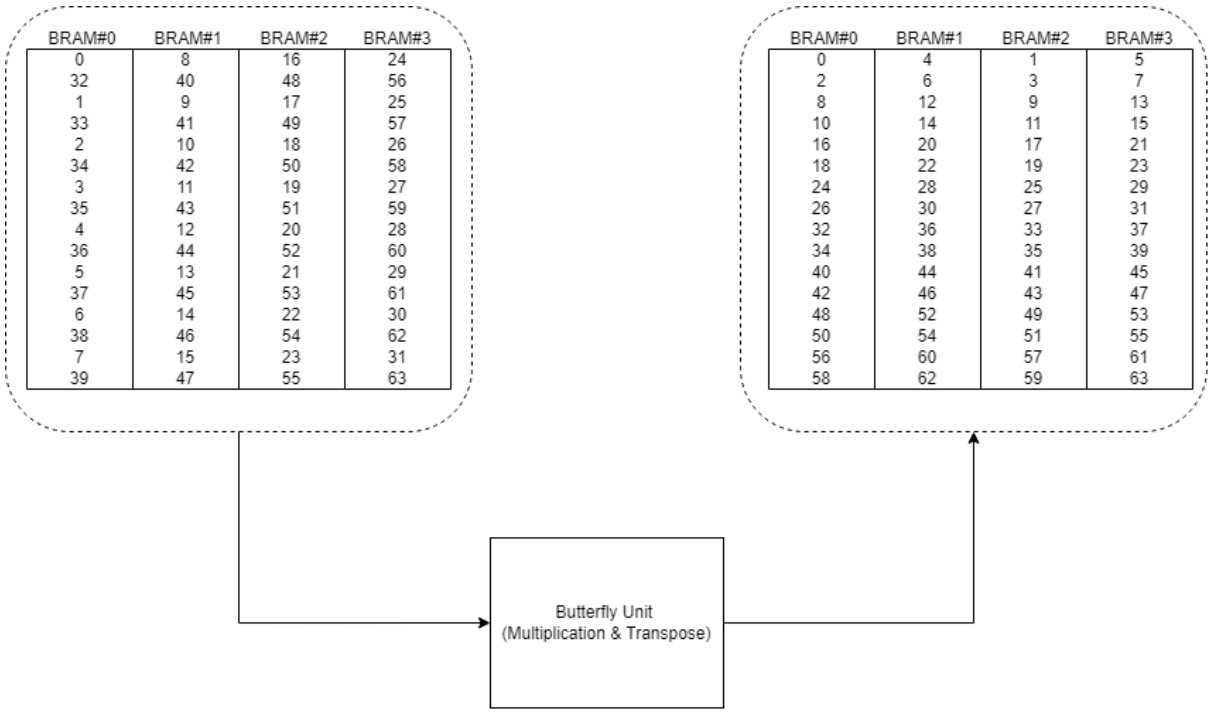

Figure 3.7 Four Step NTT Write Access Example

In these formulas, "*newID*" designates the BRAM group to be written to, while "*ID*" represents the BRAM group being read. Similarly, "*newADDR*" indicates the BRAM address to be written to, and "*ADDR*" signifies the BRAM address being read. As deduced from these equations, our design's multiplication section depends on the selected ring size and the number of PEs. This allows for easy design modifications, considering the performance and area trade-offs. For our 8×8 matrix example, Figure 3.7 visualizes the memory access pattern.

In implementing the CKKS operations on FPGA, the performance of NTT plays a key role. In this context, we explored two different NTT methods: merged NTT and four-step NTT. Both methods are potentially applicable for CKKS. However, based on performance tests for the ring sizes we worked with, we observed that merged NTT outperformed four-step NTT. Hence, we opted for the merged NTT method in implementing the CKKS scheme. We plan to develop and optimize the four-step NTT method for future work. Four-step NTT has the potential to exhibit superior performance, especially for higher ring sizes. Therefore, this method may be more effective in certain application scenarios and ring sizes.

3.2 Efficient Design-Time Flexible Hardware Architecture for Accelerating Homomorphic Encryption Operations of CKKS Scheme

This section introduces an efficient hardware architecture designed to accelerate the HE operations of the CKKS scheme. In recent years, with the growing popularity of HE in numerous application areas, there has been an increasing need for hardware-based solutions to speed up these operations. The presented architecture is designed to meet these demands, offering design-time flexibility and a performance-driven approach. In this section, we will delve into the details of how we efficiently implemented the foundational operations of the CKKS scheme, namely the homomorphic multiplication and key-switching operations, in hardware.

Within the CKKS framework, operations such as homomorphic multiplication, relinearization, and rescaling stand out as primary operations. For parameters defined as $n = 8192$ and $\lceil \log_2(q) \rceil = 218$, we introduce a cutting-edge hardware architecture meticulously tailored for superior performance and scalability. This architecture is notably *design-time configurable* in terms of the number of PE, which is crucial for striking an

optimal balance between performance and area constraints. The flexibility to adjust the PE count allows us to meet various performance versus area requirements, enhancing the adaptability and efficiency of the CKKS implementation within different scenarios.

3.2.1 Full RNS Variant of the CKKS Scheme

In this section, we scrutinize the homomorphic multiplication and key-switching operations based on the full RNS variant of the CKKS scheme. The examination is focused on delineating how these operations are orchestrated in the SEAL library, which was inaugurated by Microsoft in 2020. Following this introduction, we will provide the pseudocode to understand them comprehensively. This deep analysis will unveil the intricacies of the implementation phase and elucidate the operational mechanisms in a clear manner.

The full RNS variant of the CKKS scheme plays a crucial role in the efficient execution of HE operations. In traditional approaches, a single large modulus is employed for such functions. However, the RNS variant, as explained in the RNS section, advocates for utilizing multiple smaller moduli. This modification paves the way for using single-precision arithmetic instead of multi-precision arithmetic. Consequently, this approach allows for faster arithmetic operations in hardware without any delay. Additionally, this technique provides a more effective means of addressing operations with high computational intensity, such as key-switching. The RNS variant dissects these operations into independent smaller components, allowing for parallel processing. This parallelization lays the foundation for a more optimized MAP structure in hardware, enhancing overall performance and optimizing resource utilization.

For the effective implementation of the full RNS variant of the CKKS scheme, the scheme parameters chosen include the degree n of the polynomial ring, the maximum ciphertext modulus Q, and the RNS base Q_L consisting of $l+1$ small prime moduli. These selections are made considering the required security level and the maximum multiplicative depth l for the application. The RNS bases we employ are 32-bit in bit-length, encompassing 6 distinct RNS bases. Furthermore, for key-switching operations, we also incorporate a special prime $p = q_l$ that is exclusively designated for this process. This special prime is considered in the extended RNS base as $Q_L = \{q_0, ..., q_{l-1}, p\}$. The reasons and advantages for using these RNS bases and the special prime in the CKKS encryption scheme directly impact the overall performance and security of the scheme.

Algorithm 10 Homomorphic Multiplication for CKKS

Input: $ct_a = (ct_{a,0}, ct_{a,1}), ct_b = (ct_{b,0}, ct_{b,1})$
$ct_a, ct_b \in \mathcal{R}_Q^{(l-1)\times 2}, Q = \prod_{i=1}^{l-1} q_i$
Output: $res = (res_0, res_1, res_2),\ res \in \mathcal{R}_Q^{(l-1)\times 3}$

1: **for** $i = 1 \rightarrow l-1$ **do**
2: $\quad res_{0,i} \leftarrow ct_{a,0,i} \odot ct_{b,0,i}$
3: $\quad res_{1,i} \leftarrow ct_{a,0,i} \odot ct_{b,1,i} + ct_{a,1,i} \odot ct_{b,0,i}$
4: $\quad res_{2,i} \leftarrow ct_{a,1,i} \odot ct_{b,1,i}$
5: **return** res

3.2.1.1 Homomorphic Multiplication

Within the framework of the CKKS scheme, homomorphic multiplication is distinctly characterized by its utilization of a sequence of smaller moduli, denoted as q_i, which collectively constitute the RNS base. A primary advantage of this methodology in the CKKS is the facilitation of parallel computations across individual RNS moduli, significantly enhancing the operation's efficiency.

Compared to the homomorphic multiplication process intrinsic to the BFV scheme, the CKKS exhibits a noticeable advantage. The BFV scheme is renowned for its reliance on supplementary RNS bases and consequent engagement in resource-intensive base extensions. In contrast, as depicted in Algorithm 10, the CKKS refrains from using any additional RNS bases. It solely engages in coefficient-wise multiplication operations on ciphertexts.

Within the CKKS scheme, the homomorphic multiplication operation starts by taking in two ciphertexts, each consisting of two polynomials, based on the RNS arithmetic. Upon completion of the operation, a ciphertext encompassing three distinct components is derived. As illustrated in Algorithm 10, this procedure is portrayed step-by-step. The algorithm receives two ciphertexts, namely ct_a and ct_b, as input, each bearing a dual-component configuration within the $\mathcal{R}_Q^2$ space, where the modulus Q is construed as the product of the multiple moduli each represented by q_i. During the computational phase, the individual components of both ciphertexts undergo multiplication processes inter se, leading to the generation of a novel ciphertext, *res*, characterized by three separate components.

Algorithm 11 Relinearization for CKKS

Input: $ct_a, ct_b, ct_c \in \mathcal{R}_Q$
Input: $RK_0, RK_1 \in \mathcal{R}^l_{Q_L}$ *# Relinearization keys*
Output: $ct'_0, ct'_1 \in \mathcal{R}_{Q_L}$

$val_0, val_1 \leftarrow 0$
for $i = 0 \rightarrow l-1$ **do**
    $\alpha \leftarrow \mathrm{INTT}_{q_i}(ct_{c,i})$
    for $j = 0 \rightarrow l$ **do**
        $\bar{\beta} \leftarrow \mathrm{NTT}_{q_j}(\alpha)$
        $val_0 \leftarrow \bar{\beta} \odot RK_{0,i,j} \bmod q_j$
        $ct'_{0,j} \leftarrow ct'_{0,j} + val_0 \bmod q_j$
        $val_1 \leftarrow \bar{\beta} \odot RK_{1,i,j} \bmod q_j$
        $ct'_{1,j} \leftarrow ct'_{1,j} + val_1 \bmod q_j$
return ct'_0, ct'_1

3.2.1.2 Key Switching

Following the execution of the homomorphic multiplication, the derived ciphertext manifests three distinct components (i.e., polynomials in $\mathcal{R}_Q$). A subsequent operation, termed "key switching", is leveraged to get these three components down to a more manageable two components. Notably, the computational demands of key switching markedly eclipse those of the homomorphic multiplication. In particular, the NTT operation acts as a performance bottleneck point for key-switching as well. This is primarily due to the numbers of NTTs and INTTs engaged in the key-switching operation; the count is about the square of the number of the RNS bases.

The key-switching procedure within the CKKS scheme consists of two stages. The first stage is the *relinearization* process. In the CKKS scheme, the relinearization operation is employed to reduce the three-component ciphertext, obtained due to homomorphic multiplication, to a structure with two components. The purpose of this operation is to decrease the size of the ciphertext, paving the way for subsequent homomorphic operations to be executed more swiftly and efficiently. The steps required to accomplish this task are elaborated in detail in the "Relinearization for CKKS" operation presented in Algorithm 11.

In the CKKS scheme, the generation of relinearization keys is required to perform the relinearization operation. Relinearization keys consist of two components: RK_0 and RK_1. The RK_0 component is produced by *expanding a pseudorandom public seed*, while RK_1 is computed using the *secret key*. Both components, RK_0 and RK_1, are l-tuples and reside

Algorithm 12 Homomorphic Rescale for CKKS

Input: $ct'_0, ct'_1 \in \mathcal{R}_{Q_L}$
Output: $\bar{ct}_0, \bar{ct}_1 \mathcal{R}_Q$

$\alpha \leftarrow \text{INTT}(ct'_{0,l})$
$\beta \leftarrow \text{INTT}(ct'_{1,l})$
$halfmod \leftarrow (q_l \gg 1)$
$\bar{\alpha} \leftarrow \alpha + halfmod \mod q_l$
$\bar{\beta} \leftarrow \beta + halfmod \mod q_l$
for $i = 0 \rightarrow l-1$ **do**
    $\alpha' \leftarrow \bar{\alpha} - halfmod \mod q_i$
    $\beta' \leftarrow \bar{\beta} - halfmod \mod q_i$
    $\tilde{\alpha} \leftarrow \text{NTT}(\alpha')$
    $\tilde{\beta} \leftarrow \text{NTT}(\beta')$
    $\gamma_0 \leftarrow ct'_{0,i} - \tilde{\alpha}$
    $\gamma_1 \leftarrow ct'_{1,i} - \tilde{\beta}$
    $\bar{ct}_{0,i} \leftarrow ct_{0,i} + q_l^{-1} \odot \gamma_0 \mod q_i$
    $\bar{ct}_{1,i} \leftarrow ct_{1,i} + q_l^{-1} \odot \gamma_1 \mod q_i$
return $\bar{ct}_0, \bar{ct}_1$

in $\mathcal{R}_{Q_L}$. The notation $RK_{0,i}$ is used to select the i-th element from the l-tuple. This element is found within $\mathcal{R}_{Q_L}$ and, thus, comprises $l+1$ residue polynomials in the RNS representation. We utilize the notation $RK_{0,i,j}$ to represent the j-th residue polynomial of $RK_{0,i}$. In summary, each of RK_0 and RK_1 is a vector containing $l(l+1)$ residue polynomials.

Algorithm 11 starts with the input components ct_a, ct_b, and ct_c, which are the three components of the ciphertext obtained after the homomoprhic multiplication operation. The variables val_0 and val_1 are initialized to zero. The two nested loops iterate across all RNS bases. The outer loop applies the INTT operation for each RNS base of the third component. Meanwhile, the inner loop transforms the output of the INTT using NTT and subsequently multiplies it with the relinearization keys RK_0 and RK_1 to compute the new ciphertext components. The procedure ensures that the ciphertext components are transformed into a streamlined two-component structure, facilitating subsequent computations.

The second step is the *rescale* operation. The primary purpose of the rescale process is to scale down the ciphertext components from $\mathcal{R}_{Q_L}$ to $\mathcal{R}_Q$.This is done to obtain the same homomorphic ciphertext value in the new RNS base. The method presented in Algorithm 12 details the steps of the rescale operation.

The algorithm starts by taking ct'_0 and ct'_1 components as input. Initially, the INTT

operation is applied to the ciphertext components of the last RNS base. The results obtained are then converted into new components by adjusting them with the value of *halfmod*. Here, *halfmod* represents the right-shifted value (by one bit) of the last RNS base(p).

These newly obtained components are scaled across all RNS bases in subsequent steps. This is achieved by applying the NTT operation for each base and making corrections using previously calculated values. As a result, these processes produce the scaled components of the ciphertext.

This scaling operation aids other processes within the CKKS scheme to operate more efficiently. Specifically, this procedure enhances the overall performance and computational speed, expanding the ability to perform more operations on the ciphertext.

3.2.2 Homomorphic Multiplication and Key Switching Architecture

As described in Algorithm 10, the homomorphic multiplication in the CKKS scheme fundamentally involves modular multiplications performed point-wise in the NTT domain, as well as modular addition operations. To execute these base operations, we utilize the unified butterfly unit as a PE, detailed in Section 3.1.1. Our design is configurable, allowing variations in the number of unified butterfly units. In this section, we will present examples of our design implemented with 96 PEs and discuss how our design adapts to changes in the number of these units. The overall structure of our design is visualized in Figure 3.8.

For Homomorphic Multiplication, four ciphertext polynomials are utilized as input. Each polynomial comes with 6 RNS bases and has 8192 coefficients. To carry out modular operations, processing element groups (PE groups) are employed. There are 12 PE groups in total, with each group comprising 8 PE elements. This means there are 96 PE elements altogether. For each RNS base, two PE groups are designated, leading to 16 PE elements in total. Since Homomorphic Multiplication mainly involves point-wise multiplication and addition, operations within each RNS base are parallelized using 2 PE groups.

The capacity to read only one data from each BRAM group exists in a single clock cycle. With the aim to boost design performance by incorporating more PEs, each of the 96 PE elements is required to read two inputs simultaneously. As a result, the number of BRAM groups is double that of the PEs, leading to 192. Each BRAM group has been

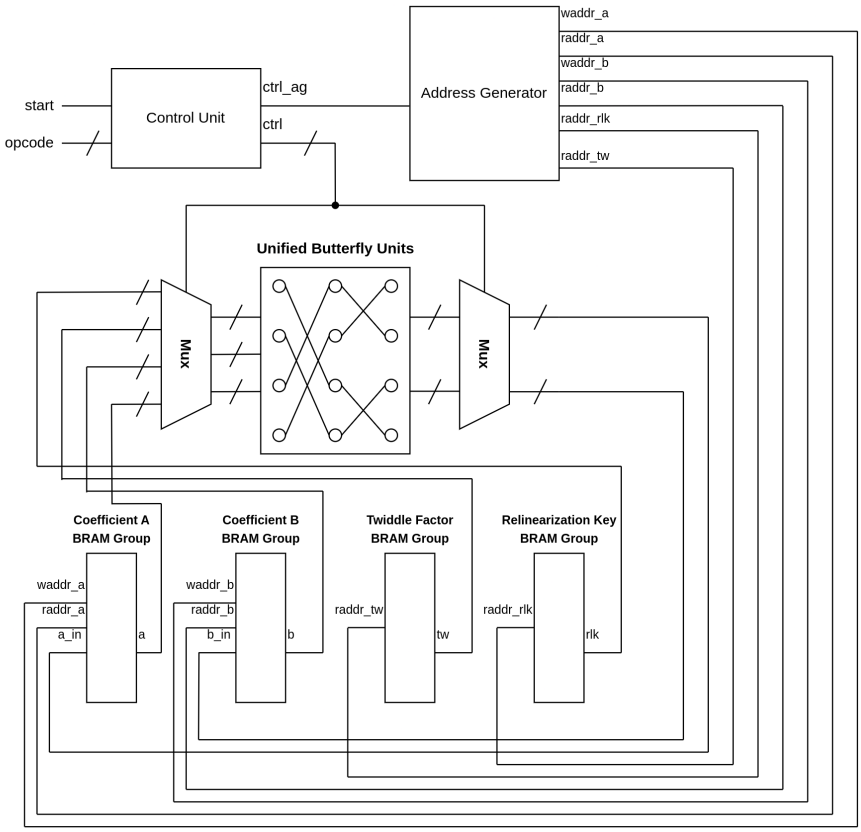

Figure 3.8 Overview of the Accelerator Architecture

fashioned with a depth of 1024 to accommodate input polynomial coefficients.

However, in Algorithm 10, to store the intermediate multiplication results of $ct_{a,0,i} \odot ct_{b,1,i}$ and $ct_{a,1,i} \odot ct_{b,0,i}$, the depth of half of the utilized BRAM groups has been increased to 2048. These intermediate multiplication values are stored between addresses 1024 and 2048 within this enhanced depth. In the end, 18 polynomials (i.e., res_0, res_1, res_2, each with 6 RNS bases) with a degree of 8192 are written back to the same BRAM groups.

In the CKKS scheme, the relinearization and rescaling operations contain fundamental arithmetic operations such as NTT, INTT, modular multiplication, addition, and subtraction. To execute the NTT and INTT functions, the merged NTT hardware, comprehensively detailed in Section 3.1.2, is employed. Owing to the configurable nature of this specific hardware to accommodate different PE counts, relinearization and rescaling operations are implemented on the FPGA using varying numbers of PEs. These processes are positioned immediately following the homomorphic multiplication operation.

A crucial point to note is that our method is specifically optimized for a particular set of parameters. Our approach is limited to applications requiring only a single multiplica-

tion depth. This characteristic affects its flexibility and the range of applications it can serve. While our design offers specific efficiencies for which it is optimized, it lacks the adaptability provided by broader designs.

While the homomorphic multiplication utilizes 96 unified butterfly units to facilitate the parallel execution of data-independent tasks, identical units have also been engaged for the relinearization and rescaling tasks. It is imperative to highlight that 408 BRAM groups are used during these sequences. In optimizing storage and access efficiencies, configurations are set with 144 BRAM groups at a depth of 2048, 56 at 2559, and 208 at 1024. Various requirements drive this specific allocation: 56 BRAM groups, at a depth 2559, are dedicated solely to storing twiddle factors. Meanwhile, 48 groups at 2048-depth are reserved for the relinearization keys. All remaining BRAM groups accommodate the input, output, and intermediary values.

The relinearization operation of the CKKS scheme is presented in Algorithm 11. During the key-switching process, each polynomial needs to be multiplied by a relinearization key, and $2l(l+1)$ relinearization keys must be stored. As $l=6$, a total of 84 relinearization keys are stored for the selected parameters. BRAMs are used to store 6 relinearization keys, and the same BRAMs are reused to store other relinearization keys. To store relinearization keys, 48 BRAM groups with a depth of 2048 are fully utilized. The BRAM groups can store relinearization keys for 6 polynomials, and the operations of 6 polynomials can be executed in parallel. The relinearization keys required for subsequent polynomials are stored sequentially in the same BRAMs. The primitive root of unity powers required for NTT and INTT operations are stored in 56 BRAMs with a depth of 2559. Due to the use of the word-level Montgomery reduction algorithm, the powers of the primitive root of unity and the relinearization keys are multiplied by the Montgomery constant R before being sent to the FPGA.

In the CKKS scheme, initiating the relinearization procedure involves executing INTT and NTT operations within two nested loops (see Algorithm 11). Through the loop unrolling technique, these operations can be executed in parallel. Eight distinct unified butterfly units are deployed to facilitate parallelization for every RNS base. Every unified butterfly unit is linked to two BRAM groups for polynomial storage and another BRAM group for holding twiddle factors. A dedicated address generation controller generates the requisite read and write addresses for the BRAM groups.

PEs are utilized for the modular multiplication of ciphertexts with relinearization keys. When executing modular multiplication operations, reading addresses ranging from 0 to 1023 are generated to read polynomial terms that are to be fed as inputs to the unified

butterfly units. The outputs from the modular multiplication performed by the unified butterfly units are recorded in the previously uninitialized 96 BRAM groups with a depth of 1024. Following these procedures, the unified butterfly units come into play again for the modular addition of the resulting polynomials. During the modular addition tasks, addresses are produced to retrieve polynomial terms, which will be fed into the unified butterfly units. The subsequent outputs are written into the 108 BRAM groups that are initialized before, each with a depth of 1024.

After completing the relinearization section, the rescaling operation, as described in Algorithm 12, is applied. The outputs from the INTT operation and the addition of half of the final RNS base (i.e., p) are saved back to the BRAM groups, from where the inputs are sourced. The unified butterfly units are configured for the INTT and addition processes. Modular subtraction, NTT, and modular addition operations are all executed in parallel using 96 unified butterfly units. The configuration of the unified butterfly units for modular subtraction is carried out in the same manner as the previously mentioned modular addition configuration. Ultimately, the produced outputs are written back to the BRAM groups where the ciphertexts, used as inputs at the onset of the relinearization process, are stored initially.

4. RESULTS AND COMPARISON

In this chapter, we present a detailed exploration of the results obtained from our proposed NTT implementations and subsequently offer a comparison with other existing designs proposed in the literature. The rapid evolution in the domain of HE and the continual demands for faster and more efficient algorithms and/or their implementations make the juxtaposition of results a crucial step to gain insight the merit of each design. Herein, we first delve deep into the outcomes from our merged and four-step NTT implementations. Additionally, the efficiency of our design in facilitating various HE operations is also elaborated upon. In the latter part of this chapter, a rigorous comparison, both in terms of resource utilization and performance metrics, is presented, reinforcing the novelties and advantages of our approach.

4.1 Implementation Results

In this section, we present the implementation results of the architecture coded in Verilog and synthesized using the Xilinx Vivado 2019.2 tool, targeting the Xilinx Alveo U280 board.

4.1.1 NTT Implementation Results

Our NTT design was successfully synthesized, achieving an operating frequency of 181 MHz. The design is tailored for $n = 8192$ to facilitate HE operations under the CKKS scheme. For investigative purposes, we also synthesized individual NTT architectures with support for $n = 4096$ and $n = 16384$, though the latter two dimensions are not used to implement the CKKS scheme. The working word size for these configurations corresponds to $\log_2 q = 32$-bit, as summarized in Table 4.1.

		Resource Utilization					Performance Metrics		
# of PE	n	**LUT**	**BRAM**	**DSP**	**LUTRAM**	**FF**	**Period (ns)**	**# of Clock Cycles**	**Latency (μs)**
8	2^{12}	7390	24	56	542	5549	5.5	3096	17
	2^{13}	7099	40	56	544	5599	5.5	6682	36.7
	2^{14}	7428	68	56	546	5653	5.5	14364	79
16	2^{12}	15201	32	112	1073	11830	5.5	1560	8.5
	2^{13}	14695	48	112	1075	11839	5.5	3354	18.4
	2^{14}	15227	80	112	1077	11915	5.5	7196	39.5
32	2^{12}	29760	48	224	2088	21406	5.5	792	4.3
	2^{13}	29189	64	224	2100	21443	5.5	1690	9.2
	2^{14}	29100	96	224	2098	21556	5.5	3612	19.8

Table 4.1 Results for the Merged NTT.

Based on the data in Table 4.1, an increase in the number of PEs results in a rise in resource utilization. For the same number of PEs, as the value of n increases, there is a noticeable growth in BRAM usage. Even though the period remains constant at 5.5 ns for all configurations, an increase in the number of PEs causes a reduction in the clock cycle count, leading to a decrease in latency. This suggests that designs with more PEs have a faster data processing capability. In conclusion, adjusting the number of PEs on different FPGA boards makes it possible to strike a balance between resource usage and performance.

Our four-step NTT design for a ring size of 2^{12} was synthesized on the FPGA using 24 PEs. With this configuration, the design operates at 5.5 ns and consumes a total of 1983 LUTs, 142 FFs, and 168 DSPs. For the 24 PEs, the total number of clock cycles was determined to be 4396, which corresponds to a latency of 24.1 μs and a throughput of 10.6 μs.

When examining the clock cycle variations for configurations with 48 and 96 PEs concerning the four-step NTT:

- For the configuration with 48 PEs, the total number of clock cycles is found to be 2610, which results in a latency of 14.3 μs and a throughput of 6.4 μs.

- For the configuration with 96 PEs, the total number of clock cycles is found to be 1688, which corresponds to a latency of 9.3 μs and a throughput of 4.3 μs.

These clock cycle duration figures indicate that the total required clock cycle count decreases as the number of PEs increases. This suggests that the data processing speed increases with more PEs in use.

In our FPGA design, when we compare the latency of the Four-Step NTT with that of the Merged NTT, we observe that the Four-Step NTT takes longer. This is due to the additional multiplication operation in the Four-Step NTT, which results in more clock cycles. Therefore, we opted for the Merged NTT method when implementing the CKKS operations.

However, we must recognize the advantages brought by the pipelined structure of the Four-Step NTT. High throughput is achieved, especially when using an equal number of PEs for each pipeline stage. Furthermore, when the data size exceeds the capacity of the BRAMs, it's essential to consider the advantageous memory access pattern offered by the Four-Step NTT when reading data from off-chip memory.

4.1.2 Homomorphic Encryption Operations Implementation Results

The proposed architecture provides support for homomorphic multiplication, relinearization, and rescaling operations in line with the CKKS scheme. In this study, the proposed architecture, based on the Xilinx Alveo U280 FPGA platform, employs 372940 LUTs, 672 DSPs, and 768 BRAMs within a structure, where the PE count is 96 (for the other two potential number of PEs, see Table 4.2). The results presented in Table 4.2 contain the collective performance of homomorphic multiplication, relinearization, and rescaling operations, reflecting the integrated outcome of these processes. These operations are executed in 2581, 72175, and 18591 clock cycles. It is worth noting that these clock cycle figures do not include the write durations to the BRAM components from external sources.

For the PE counts, two distinct selections, $\#\text{PE} \in \{96, 192\}$, are found to be suitable in terms of Xilinx Alveo U280 FPGA resource utilization. However, the required LUT value for $\#\text{PE} = 384$ exceeds the capacity of the Xilinx Alveo U280 FPGA. The detailed synthesis results are presented in Table 4.2.

Our hardware-oriented approach, with $\#\text{PE} = 96$, provides a 15-fold speedup in the

		Resource Utilization				Performance Metrics		
# of PE	n	**LUT**	**BRAM**	**DSP**	**FF**	**Period (ns)**	**Clock Cycle**	**Latency (μs)**
96	2^{13}	372940	768	672	9826	5.5	93350	513.4
192	2^{13}	754128	928	1344	264399	5.5	46886	257.8
384	2^{13}	1520173	1632	2688	520961	5.5	23846	131.1

Table 4.2 Results for the CKKS Operations (Multiplication + Relinearization + Rescaling)

homomorphic multiplication operation and a 4-fold speedup in the key switch operation compared to the software implementation based on the Microsoft SEAL library running on an AMD Ryzen 7 3800x CPU.

Our design can be easily modified for FPGA platforms with higher resource capacities. Increasing the number of PEs can further enhance performance metrics. As mentioned in Section 3.2, making these adaptations is straightforward. Therefore, while our current results are based on the Xilinx Alveo U280 FPGA, there is significant potential for improving performance and efficiency on more advanced FPGA platforms.

4.2 Comparison

In this section, we present comparison of our implementation with those in the literature, first for NTT operation, then for homomorphic operations.

4.2.1 NTT Comparison

Table 4.3 presents a comprehensive comparison of NTT implementations. The comparison is divided into two main categories: resource utilization and performance metrics. Resource utilization includes the usage of Lookup Tables (LUTs), Block RAMs (BRAMs), and Digital Signal Processing (DSP) blocks, while performance metrics contain the processing time (Period) and latency (Latency). A significant portion of the works (Ozturk et al. (2017), Sinha Roy et al. (2019), Matteo et al. (2023), Duong-Ngoc et al. (2022), Paludo and Sousa (2022), Xin et al. (2021)) focus on developing hardware accelerators

for homomorphic encryption. Some studies, such as Su et al. (2022) and Ye et al. (2022), propose reconfigurable architectures and pipelined structures to accelerate polynomial multiplication. The work of Kawamura et al. (2018) concentrates on loop structure optimization, targeting the acceleration of the NTT. These contributions take significant steps in accelerating computations in HE.

The Kawamura et al. (2018) study utilizes High-Level Synthesis (HLS) tools to optimize the loop structure in the software definition of the NTT operation through loop flattening and reduction of iteration counts, leading to hardware-level improvements on an FPGA. These techniques maximize the FPGA's parallel processing capabilities and significantly improve computational efficiency. In addition, our design, featuring a $\#\text{PE} = 32$ architecture, has achieved significantly better results in performance metrics, particularly in terms of latency. More importantly, our $\#\text{PE} = 16$ and $\#\text{PE} = 8$ architectures offer superior results compared to other studies in both performance metrics and the usage of LUTs and BRAMs. These outcomes demonstrate that our design strikes an excellent balance between resource efficiency and computational performance.

When comparing the performance and resource utilization of the studies, the works of Matteo et al. (2023) and Paludo and Sousa (2022) offer exceptionally economical solutions regarding resource usage. The study of Matteo et al. (2023) addresses the issue of HE libraries such as Microsoft SEAL not being designed for resource-constrained platforms by proposing a hardware accelerator designed explicitly for a HE library for embedded platforms, SEAL-Embedded. This accelerator includes a memory architecture that reduces I/O latency and a dedicated module for the generation of roots of unity. On the other hand, the Paludo and Sousa (2022) study proposes an architecture for the acceleration of NTT using a Montgomery-based butterfly. Compared to our work, the Matteo et al. (2023) and Paludo and Sousa (2022) studies have lower resource consumption regarding LUT, BRAM, and DSP usage. However, when it comes to latency, our study supports lower latency values than both Matteo et al. (2023) and Paludo and Sousa (2022). For instance, while performing operations at a size of 2^{12}, the Matteo et al. (2023) study offers a latency of 136.58 μs and the Paludo and Sousa (2022) study offers 11 μs. Our study achieves a significant speed advantage with only 4.3 μs of latency. This reduced latency is particularly advantageous for time-critical applications and, despite higher resource usage, presents a preferable solution where latency is critical.

The dedicated hardware accelerator focused on in the Ozturk et al. (2017) study is optimized for somewhat homomorphic encryption-based schemes, offering a structure that can multiply large polynomials much faster. However, there are differences between our work and Ozturk et al. (2017) in terms of resource utilization and performance metrics.

Regarding resource utilization, the Ozturk et al. (2017) study consumes 219K LUTs and 768 DSPs, and when looking at performance metrics, it achieves a latency of 24.5 μs for operations of size 2^{14}. In comparison, our NTT implementation uses 29100 LUTs and 224 DSPs for operations of the same size and offers a latency of 19.8 μs. It is necessary to note that the resource utilization data for the Ozturk et al. (2017) study is for the polynomial multiplier. It should be noted that the data presented in our study, including the resource utilization figures, specifically pertains to the NTT implementation. However, the performance metrics, i.e., the latency times, are for NTT operations in both studies. In this context, our study offers an advantage in efficiency and optimization, providing better latency times with lower resource usage.

		Resource Utilization			**Performance Metrics**	
WORK	n	**LUT**	**BRAM**	**DSP**	**Period (ns)**	**Latency (μs)**
Ozturk et al. (2017)	2^{14}	219K	193	768	4	24.5
Sinha Roy et al. (2019)	2^{12}	64K	400	200	4.4	73
Matteo et al. (2023)	2^{12}	3320	29.5	42	5.5	136.58
Duong-Ngoc et al. (2022)	2^{16}	149K	25	564	5	2684
Paludo and Sousa (2022)	2^{12}	7.9K	24	32	3.6	11
Su et al. (2022)	2^{12}	14K	79	80	4	12.3
Ye et al. (2022)	2^{12}	17K	24	286	6.6	55
Xin et al. (2021)	2^{12}	176K	550	1344	6.6	20
Kawamura et al. (2018)	2^{12}	30K	36	12	10	333
	2^{13}	36K	82	13	10	703
	2^{14}	41K	183	26	10	1482
Our work	2^{12}	29760	48	224	5.5	4.3
	2^{13}	29189	64	224	5.5	9.2
	2^{14}	29100	96	224	5.5	19.8

Table 4.3 Comparative Table for Merged NTT Implementations.

The resource utilization and performance metrics of our study are competitive when compared to similar works in the literature. For instance, Sinha Roy et al. (2019)'s work utilizes 64K LUTs and 200 DSPs to achieve a latency of 73 μs for operations of size 2^{12}. In contrast, our study only requires 29760 LUTs and 224 DSPs to achieve a significantly lower latency of 4.3 μs for the same operation size. However, comparing our results with those of Duong-Ngoc et al. (2022), which focuses on operations of size 2^{16}, may only partially be fair due to the different ring sizes addressed. Nonetheless, the algorithmic and architectural features of our NTT implementation possess the flexibility and capacity to be adapted to operations of size 2^{16}. Preliminary results suggest that we could significantly improve the resource utilization and latency metrics presented by Duong-Ngoc et al. (2022). Moreover, while the works of Su et al. (2022) and Ye et al. (2022) also exhibit higher resource usage and latency for similar operation sizes, Xin et al.

(2021)'s work, despite being closer to our study in terms of resource utilization, still falls short with a latency of 20 μs compared to our result of 4.3 μs.

In conclusion, the design developed in this study offers competitive latency performance. Furthermore, the reconfigurability of our design for different ring sizes and PE counts demonstrates its adaptability across a broad spectrum of applications. Comparative analyses of performance metrics and resource utilization reveal that our design not only provides a significant advantage in latency but is also competitive in resource utilization. With its low latency times for small and large ring sizes, our design emerges as an effective hardware acceleration solution for HE.

4.2.2 Homomorphic Encryption Operations Comparison

Table 4.4 compares the resource utilization and performance metrics for HE operations on different FPGA devices. The study Mert et al. (2022) represents an application with a 2^{14} parameter on the Alveo U250 FPGA device, demonstrating a significant resource consumption with 1,093,250 ALM/LUT, 1,576.5 BRAM, 3,607 DSP, and 931 URAM. This application operates with a period of 5ns and a delay of 669.39 microseconds.

			Resource Utilization				Performance Metrics	
WORK	**FPGA Device**	n	**ALM/LUT**	**BRAM**	**DSP**	**URAM**	**Period (ns)**	**Latency (μs)**
Mert et al. (2022)	U250	2^{14}	1093250	1576.5	3607	931	5	497.24
Riazi et al. (2020)	Stratix10	2^{13}	698884	10340	2610	-	3.3	-
Our	U280	2^{13}	754128	928	1344	-	5.5	257.8

Table 4.4 Comparison Table for Homomorphic Operations: Multiplication + Key Switching.

Mert et al. (2022) focuses on the flexibility needs of HE applications. This flexibility reflects the varying multiplication depth requirements of different homomorphic applications. Notably, these applications' different multiplication depths result in variability in the parameter sets used. In contrast, our application is optimized according to a specific parameter set, limiting it to a single multiplication depth. Consequently, our method needs more flexibility in Mert et al. (2022). While Mert et al. (2022) is designed to adapt to the needs of various HE applications with varying multiplication depths, our approach can only be employed with a fixed parameter set, inherently constraining it to applications requiring just one multiplication. The study Mert et al. (2022) introduces the Residue Polynomial Arithmetic Unit (RPAU) design to address this need for flexibility.

This RPAU comprises a NTT unit for polynomial multiplication operations, two parallel dyadic arithmetic units, and customized on-chip memory for result residue polynomials. This design enables operations to be executed with the speed and efficiency required to meet the demands of HE applications.

Mert et al. (2022) performed with parameter sets 2^{14} and 2^{15} for the U250 FPGA device. Our application's ring size parameter is closer to MEDAH's 2^{14} parameter set, and thus we consider this set for our comparisons. Given that MEDAH is designed for a larger ring size at 2^{14}, this higher parameter set is anticipated to have greater resource consumption and more latency. Unlike other studies, Mert et al. (2022) utilized 931 URAMs. Mert et al. (2022) opts for URAMs to store RPMs efficiently and also to store the key-switching key. This usage facilitates more optimized memory access based on data dependencies and access patterns.

Riazi et al. (2020) provides the results of another significant research that examines RNS-based HE operations on a Stratix10 FPGA device. This study was conducted with a parameter set similar to ours. There are distinct differences in design methodologies between Riazi et al. (2020) and our approach. The Riazi et al. (2020) study aims for high throughput by partitioning key-switching operations into stages and employing a separate block for each stage. Due to the focus on maximizing throughput with the block-pipeline architecture, Riazi et al. (2020) does not emphasize latency. Consequently, the reason Riazi et al. (2020) does not report latency times is due to these structural features of the design. Additionally, Riazi et al. (2020) can perform 22,536 operations (consisting of both multiplication and key-switching operations) per second.

However, in our approach, programmability is prioritized with a specific instruction set architecture, and PEs and BRAMs are repeatedly used to compute different steps of homomorphic operations. Naturally, our approach embodies a low-latency-focused architecture. As a result, the design introduced in Riazi et al. (2020) can perform approximately 5 times more operations per second than ours.

5. CONCLUSION AND FUTURE WORK

This chapter provides an overview of the studies covered in this dissertation, concluding with potential avenues for future research.

5.1 Conclusion

In this thesis, we first proposed a parametric merged-NTT hardware to accelerate the homomorphic operations of the CKKS scheme. From prior works, we observed that there was generally a focus on specific ring sizes and the number of PEs. This implies a need for more flexibility in existing studies to cater to different polynomial sizes or PE configurations. In our work, we introduced a design-time configurable, parametric integrated NTT hardware architecture that can respond to various ring sizes, hardware resource constraints, and numbers of PEs. This versatile structure allows the architecture to be used for various hardware environments, polynomial sizes, and numbers of PEs.

Furthermore, we presented a hardware architecture optimized for NTT operations that offers design-time flexibility. Our proposed hardware provides a flexible structure, taking into account hardware resource constraints. This design can perform the NTT operation without any stalls between NTT stages. The suggested merged-NTT architecture is applied to the key-switching and homomorphic multiplication operations of the CKKS scheme, providing design-time reconfigurability that allows for an increase in the number of PEs.

In our thesis, we also proposed a design-time flexible hardware architecture for the four-stage NTT method. This alternative NTT algorithm presents another potential approach for rapid NTT computation. Especially for larger ring sizes, there are potential advantages to the four-stage NTT. Our thesis discussed how this method could be more effective for larger ring sizes and its potential advantages for specific application scenarios.

Within the scope of this thesis, the presented architecture possesses a detailed structure supporting the CKKS HE scheme. It can efficiently execute critical operations such as homomorphic multiplication, relinearization, and rescaling. Implemented on the Xilinx Alveo U280 FPGA platform, this architecture optimizes resource utilization based on selected PE values, offering a fast and resource-efficient solution. Moreover, our hardware-centric approach provides significant speed enhancements compared to software solutions, particularly those based on the Microsoft SEAL library. This suggests that FPGAs might be a potentially more suitable platform for HE operations. The customizable nature of the presented architecture is designed to be flexible enough to be compatible with different FPGA platforms, implying easy adaptability to more advanced or different FPGA platforms. Lastly, this study emphasizes the importance of an FPGA-based optimized approach in HE operations theoretically and through practical applications.

5.2 Future Work

HE schemes, especially the CKKS scheme, are witnessing growing interest due to their potential for secure and efficient computations on encrypted data. Our current design presents a foundational step in implementing CKKS on FPGA platforms, emphasizing low latency. However, there are further improvements and paths to be explored. In this context, we highlight the potential directions we plan to pursue in our ongoing research.

Our emphasis on low latency in accelerating the CKKS scheme has shown that we need to balance the low-latency design we implemented in our NTT and the efficiency of the CKKS key-switching algorithm. While improving the key-switching mechanism can enhance the overall efficiency of the CKKS scheme, especially in real-time applications where timely execution of homomorphic operations is critical, it is imperative that we concurrently optimize the associated efficiency.

Our parametrically merged-NTT hardware, which primarily focuses on latency improve-

ment for our NTT and homomorphic operations on FPGA implementations, can integrate with other HE schemes. By leveraging the high throughput characteristics of the four-step NTT, it is feasible to construct a pipeline structure that offers increased throughput for homomorphic operations. This integration can unveil extensive use cases for the hardware, especially with throughput-optimized designs utilizing the four-step NTT that have great potential to explore. We aim for FPGA platforms to address a wide range of cryptographic needs, which are closely related to the adaptability of the hardware.

Establishing the right balance between encryption security, performance, and efficiency necessitates allowing the CKKS scheme to operate at different ring sizes. While our current design offers a specific range, we anticipate challenges associated with higher ring sizes, particularly concerning FPGA memory constraints. When dealing with larger ring sizes that exceed onboard memory, solutions such as the 4-step NTT, recognized for its memory efficiency, can be considered.

Energy efficiency becomes paramount as cryptographic hardware finds increased applications, especially in mobile and IoT devices. Our current design underscores the need for energy optimization. Future iterations should emphasize both computational efficiency and reduced energy consumption. By adopting energy-efficient algorithms and architectural optimizations, we can pave the way for a sustainable and efficient future in cryptographic hardware design on FPGA platforms.

HE is a crucial tool for secure and efficient data processing. By implementing the CKKS scheme on FPGA platforms, we strive to maximize this potential. Despite our current design's advantages, some areas need improvements to push further. In the coming period, we plan to enact these enhancements to boost the overall performance of our cryptographic solutions.

BIBLIOGRAPHY

Acar, A., Aksu, H., Uluagac, A. S., and Conti, M. (2018). A survey on homomorphic encryption schemes: Theory and implementation. *ACM Comput. Surv.*, 51(4).

Agarwal, R. and Burrus, C. (1975). Number theoretic transforms to implement fast digital convolution. *Proceedings of the IEEE*, 63(4):550–560.

Ayduman, C., Koçer, E., Kırbıyık, S., Can Mert, A., and Savaş, E. (2023). Efficient design-time flexible hardware architecture for accelerating homomorphic encryption. In *2023 IFIP/IEEE 31st International Conference on Very Large Scale Integration (VLSI-SoC)*, pages 1–7.

Aysu, A., Patterson, C., and Schaumont, P. (2013). Low-cost and area-efficient fpga implementations of lattice-based cryptography. In *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pages 81–86.

Barrett, P. (1986). Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In *Advances in Cryptology — CRYPTO' 86*, page 311–323.

Bisheh-Niasar, M., Azarderakhsh, R., and Mozaffari-Kermani, M. (2021). High-speed ntt-based polynomial multiplication accelerator for crystals-kyber post-quantum cryptography. Cryptology ePrint Archive, Paper 2021/563. https://eprint.iacr.org/2021/563.

Brakerski, Z., Gentry, C., and Vaikuntanathan, V. (2011). Fully homomorphic encryption without bootstrapping. Cryptology ePrint Archive, Paper 2011/277. https://eprint.iacr.org/2011/277.

Cheon, J. e. a. (2017). Homomorphic encryption for arithmetic of approximate numbers. In *Advances in Cryptology – ASIACRYPT 2017*, pages 409–437.

Chu, E. and George, A. (1999). *Inside the FFT black box: serial and parallel fast Fourier transform algorithms*. CRC press.

Cooley, J. W. and Tukey, J. W. (1965). An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301.

Dai, W. and Sunar, B. (2015). cuhe: A homomorphic encryption accelerator library. Cryptology ePrint Archive, Paper 2015/818. https://eprint.iacr.org/2015/818.

Duong-Ngoc, P., Kwon, S., Yoo, D., and Lee, H. (2022). Area-efficient number theoretic transform architecture for homomorphic encryption. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 70(3):1270–1283.

Gentry, C. (2009). Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*.

Huang, J., Zhang, J., Zhao, H., Liu, Z., Cheung, R. C. C., Koç, . K., and Chen, D. (2022). Improved plantard arithmetic for lattice-based cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(4):614–636.

Karatsuba, A. A. and Ofman, Y. P. (1962). Multiplication of many-digital numbers by automatic computers. In *Doklady Akademii Nauk*, volume 145, pages 293–294. Russian Academy of Sciences.

Kawamura, K., Yanagisawa, M., and Togawa, N. (2018). A loop structure optimization targeting high-level synthesis of fast number theoretic transform. In *2018 19th International Symposium on Quality Electronic Design (ISQED)*, pages 106–111.

Li, S., Chen, Y., Chen, L., Liao, J., Kuang, C., Li, K., Liang, W., and Xiong, N. (2023). Post-quantum security: Opportunities and challenges. *Sensors*, 23(21).

Longa, P. and Naehrig, M. (2016). Speeding up the number theoretic transform for faster ideal lattice-based cryptography. In Foresti, S. and Persiano, G., editors, *Cryptology and Network Security*, pages 124–139, Cham. Springer International Publishing.

Lyubashevsky, V., Peikert, C., and Regev, O. (2013). On ideal lattices and learning with errors over rings. *J. ACM*, 60(6).

Matteo, S. D., Gerfo, M. L., and Saponara, S. (2023). Vlsi design and fpga implementation of an ntt hardware accelerator for homomorphic seal-embedded library. *IEEE Access*, 11:72498–72508.

Mert, A. C. (2021). *Efficient Hardware Implementations for Lattice-Based Cryptography Primitives*. PhD thesis, Sabancı University, İstanbul, Turkey.

Mert, A. C., Aikata, Kwon, S., Shin, Y., Yoo, D., Lee, Y., and Roy, S. S. (2022). Medha: Microcoded hardware accelerator for computing on encrypted data. Cryptology ePrint Archive, Paper 2022/480. https://eprint.iacr.org/2022/480.

Mert, A. C., Öztürk, E., and Savaş, E. (2019). Design and implementation of a fast and scalable ntt-based polynomial multiplier architecture. *2019 22nd Euromicro Conference on Digital System Design (DSD)*, pages 253–260.

Montgomery, P. L. (1985). Modular multiplication without trial division. *Mathematics of computation*, 44(170):519–521.

Ozturk, E., Doroz, Y., Savas, E., and Sunar, B. (2017). A custom accelerator for homomorphic encryption applications. *IEEE Transactions on Computers*, 66(1):3–16.

Paillier, P. (1999). Public-key cryptosystems based on composite degree residuosity classes. In Stern, J., editor, *Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238. Springer.

Paludo, R. and Sousa, L. (2022). Ntt architecture for a linux-ready risc-v fully-homomorphic encryption accelerator. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 69(7):2669–2682.

Pei, D., Salomaa, A., and Ding, C. (1996). *Chinese remainder theorem: applications in computing, coding, cryptography*. World Scientific.

Pollard, J. M. (1971). The fast fourier transform in a finite field. *Mathematics of Computation*, 25:365–374.

Pöppelmann, T., Oder, T., and Güneysu, T. (2015). High-performance ideal lattice-based cryptography on 8-bit atxmega microcontrollers. In Lauter, K. and Rodríguez-Henríquez, F., editors, *Progress in Cryptology – LATINCRYPT 2015*, pages 346–365, Cham. Springer International Publishing.

Pöppelmann, T. and Güneysu, T. (2012). Towards efficient arithmetic for lattice-based cryptography on reconfigurable hardware. pages 139–158.

Riazi, M. S., Laine, K., Pelton, B., and Dai, W. (2020). Heax: An architecture for computing on encrypted data. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1295–1309, New York, NY, USA. Association for Computing Machinery.

Roy, S. S., Vercauteren, F., Mentens, N., Chen, D. D., and Verbauwhede, I. (2014). Compact ring-lwe cryptoprocessor. In Batina, L. and Robshaw, M., editors, *Cryptographic Hardware and Embedded Systems – CHES 2014*, pages 371–391, Berlin, Heidelberg. Springer Berlin Heidelberg.

SEAL (2020). Microsoft SEAL (release 3.6). https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA.

Sinha Roy, S., Turan, F., Jarvinen, K., Vercauteren, F., and Verbauwhede, I. (2019). Fpga-based high-performance parallel architecture for homomorphic computing on encrypted data. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

Su, Y., Yang, B.-L., Yang, C., Yang, Z.-P., and Liu, Y.-W. (2022). A highly unified reconfigurable multicore architecture to speed up ntt/intt for homomorphic polynomial multiplication. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 30(8):993–1006.

Türkoğlu, E. R., Özcan, A. ., Ayduman, C., Mert, A. C., Öztürk, E., and Savaş, E. (2022). An accelerated gpu library for homomorphic encryption operations of bfv scheme. In *2022 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1155–1159.

Winkler, F. (1996). *Polynomial algorithms in computer algebra / F. Winkler*. Texts and monographs in symbolic computation. Springer, Wien.

Xin, G., Zhao, Y., and Han, J. (2021). A multi-layer parallel hardware architecture for homomorphic computation in machine learning. In *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5.

Ye, Z., Cheung, R. C. C., and Huang, K. (2022). Pipentt: A pipelined number theoretic transform architecture. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 69(10):4068–4072.

Zhang, N., Qin, Q., Yuan, H., Zhou, C., Yin, S., Wei, S., and Liu, L. (2020). Nttu: An area-efficient low-power ntt-uncoupled architecture for ntt-based multiplication. *IEEE Transactions on Computers*, 69(4):520–533.